|  |
| --- |
| **<<ADL FRAMEWORK>>** |
| 구조설계서 |
|  |
|  |
| **2021-03-25** |
| **홍덕기** |

이 문서는 **<<ADL FRAMEWORK>>** 개발을 위한 구조설계서이다.

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Description |
| 0.1 | 2021-03-25 | 홍덕기 | 초기 문서 생성 |
| 0.5 | 2021-04-04 | 홍덕기 | 최종구조 생성 |
| 1.0 | 2021-04-19 | 홍덕기 | 시스템 구조설계 완료 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1. 시스템 개요 4](#_Toc69756637)

[1.1. 비즈니스 환경 4](#_Toc69756638)

[1.2. 시스템의 정의 6](#_Toc69756639)

[2. 요구사항 7](#_Toc69756640)

[2.1. 기능적 요구사항 7](#_Toc69756641)

[2.1.1. Use Case 7](#_Toc69756642)

[2.1.2. ADL Specification 9](#_Toc69756643)

[2.2. 비기능적 요구사항품질 속성 13](#_Toc69756644)

[3. 시스템 구조 16](#_Toc69756645)

[3.1. 시스템 개요 16](#_Toc69756646)

[3.2. 전체 시스템 구조 18](#_Toc69756647)

[3.3. 주요기능 동작 설명 19](#_Toc69756648)

[3.3.1. Program 구동 시 동작 19](#_Toc69756649)

[3.3.2. Translate 동작 20](#_Toc69756650)

[3.3.3. Autocomplete 동작 24](#_Toc69756651)

[3.3.4. Code Generate 동작 26](#_Toc69756652)

[3.4. 시스템 특성 28](#_Toc69756653)

[3.4.1. Extension tier의 변경 용이성 28](#_Toc69756654)

[3.4.2. 성능 최적화 29](#_Toc69756655)

[3.4.3. 3-tier 구조로 분리되어 독립적인 개발이 가능 29](#_Toc69756656)

[4. 모듈 사양 30](#_Toc69756657)

[4.1. 모듈 상세 31](#_Toc69756658)

[4.1.1. Interface Layer – Core Layer 31](#_Toc69756659)

[4.1.2. Core Layer – Extension Layer 32](#_Toc69756660)

[4.1.3. Translate Task 34](#_Toc69756661)

[4.1.4. Autocomplete Task 36](#_Toc69756662)

[4.1.5. CodeGenerate Task 37](#_Toc69756663)

[4.2. Work Assignment 38](#_Toc69756664)

[부록 39](#_Toc69756665)

[A. 도메인 모델 43](#_Toc69756666)

[B. 품질 시나리오 47](#_Toc69756667)

[C. 품질 시나리오 분석 49](#_Toc69756668)

[D. 후보 구조 51](#_Toc69756669)

[E. 후보 구조 평가 86](#_Toc69756670)

[F. 최종 구조 설계 99](#_Toc69756671)

[G. 최종 구조 평가(ATAM) 109](#_Toc69756672)

# 시스템 개요

## 비즈니스 환경

다양한 스마트폰용 OS들이 있어왔지만, 그 중에서도 Android와 iOS는 세계의 스마트폰 시장을 양분하고 있다. 특히 Android는 iOS대비 Application을 개발하는데 있어서 개발 환경 및 배포 등에 좀 더 수월한 면이 있어 더 많은 수의 Application들이 개발되고 있다.

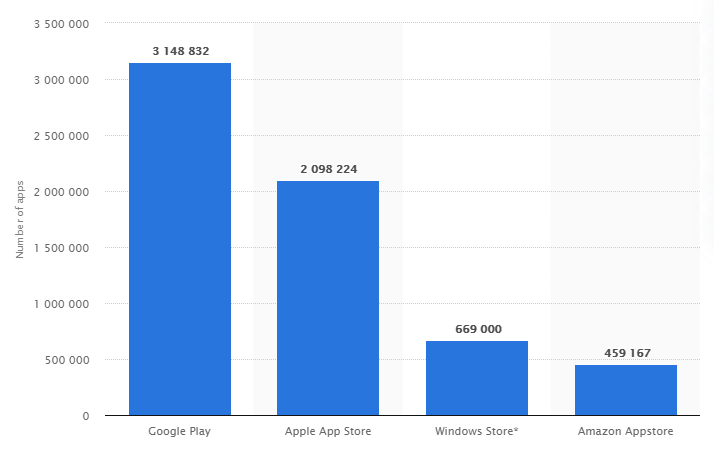


그림 . [Number of apps available in leading app stores as of 4th quarter 2020](https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores)

하지만 늘어나는 Application 수에 비해 그 품질은 높지 않아, Install된 이후에 Uninstall하는비율이 iOS대비 상당히 높은 상황이다.

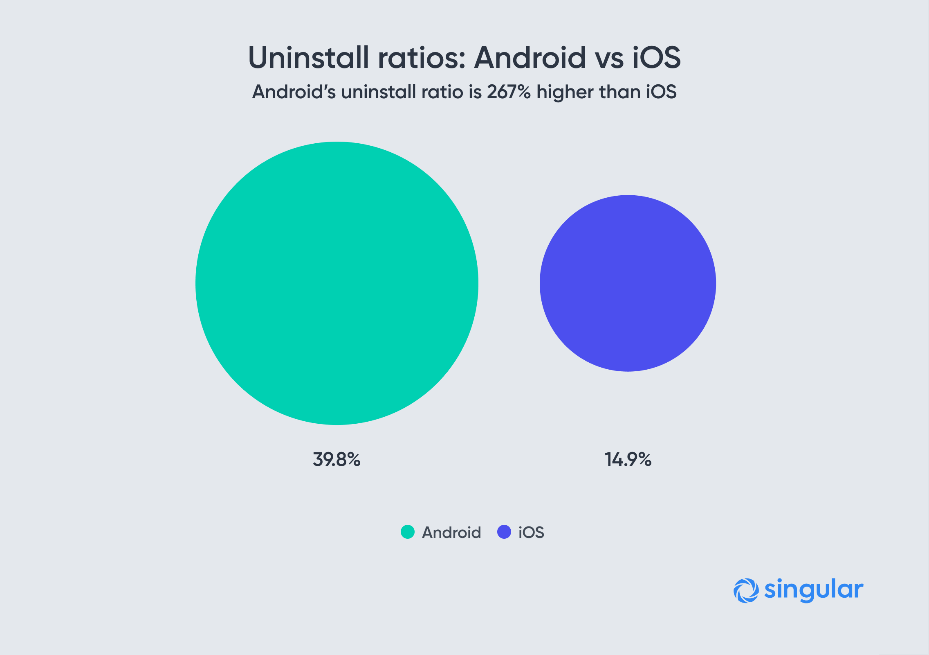


그림 . [Android uninstall rate is 2.5X more than iOS](https://www.singular.net/blog/app-uninstall-rates-and-coronavirus-lockdown-is-bad-for-retention/)

Android Application의 경쟁력을 높이는 방법은 여러가지가 있을 수 있겠지만, 그 중에서도 완성도 높은 Architecture는 Application의 용이한 update 및 각종 Quality 지표에 있어서 가장 영향을 끼치는 부분이라 할 수 있겠다.

이런 Architecture 설계는 중요성이 매우 높음에도 불구하고 현재 설계를 위한 공식 Tool이 존재하지 않고, 몇 안 되는 S/W(그림1. AcmeStudio) 들도 그 사용법이 까다롭고 Android Application Domain에 특화되어 있지 않아서 사용이 어려운 상황이다.

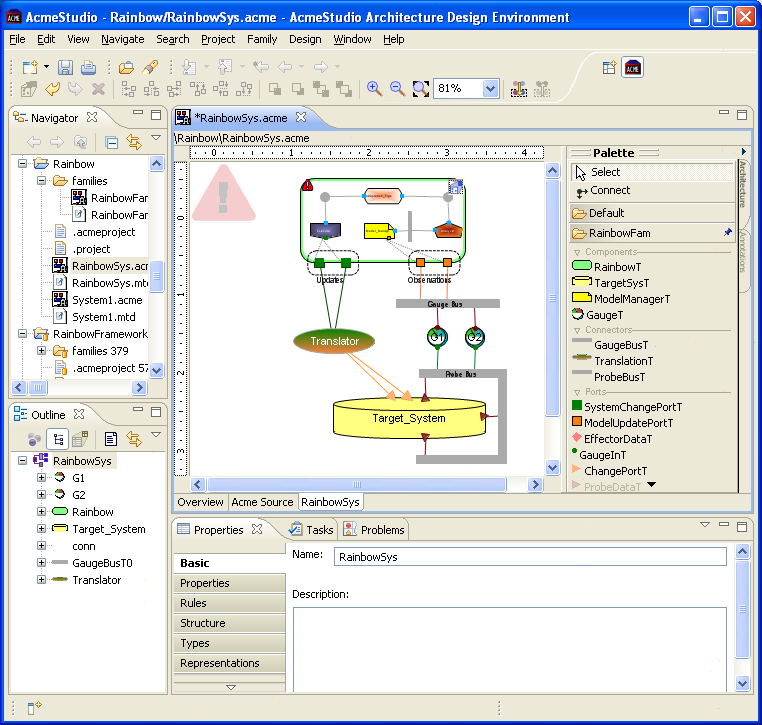


그림 . ADL 도구 예제(AcmeStudio)

이에 따라, Android Application Architecture를 위한 ADL(Architecture Description Language)을 정의하고 이를 활용하는 다양한 ADL 도구를 만들 수 있는 ADL Framework을 개발하고자 한다.

## 시스템의 정의

ADL(Architecture Description Language)은 소프트웨어 구조를 명세하기 위한 언어 또는 개념적인 모델이다. 본 시스템은 Android Application을 위한 ADL을 정의하고 해당 Language로 작성된 Code를 Graph 형태로 변환시키는 ADL Framework이다. 본 시스템을 활용해서 ADL 도구 개발자가 ADL 도구를 개발하고, 구조 설계자는 이를 활용해서 ADL Code를 작성한다. 전체 구조는 아래와 같다.

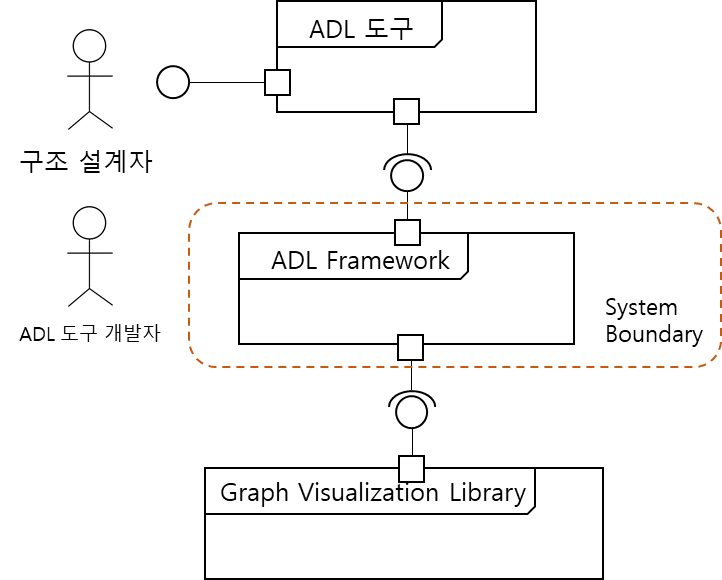


그림 . 시스템 정의

구조 설계자가 ADL 도구를 통해 본 시스템이 정의한 ADL Code를 작성하고 Translate 요청을 하면, 본 시스템은 ADL Code를 Graph Visualization Library가 인식할 수 있는 DOT([DOT (graph description language) - Wikipedia](https://en.wikipedia.org/wiki/DOT_(graph_description_language)))과 같은 Graph Description Language로 변환을 하고, 변환된 Code를 다시 Graphviz([Graphviz - Wikipedia](https://en.wikipedia.org/wiki/Graphviz))와 같은 Graph Visualization Library를 활용해서 Graph Image로 변환한 후 이를 ADL 도구로 반환한다.

# 요구사항

## 기능적 요구사항

### Use Case

ADL Framework의 Use Case는 다음과 같다.



그림 . Use Case Diagram

|  |  |
| --- | --- |
| **UC\_01** | **Translate** |
| 설명 | 작성된 ADL code를 Graph Image로 변경해서 반환한다. |
| 행위자 | ADL 도구 |
| 선행조건 |  |
| 후행조건 |  |
| 기본 동작 | 1. ADL 도구가 시스템으로 Translate를 요청한다. 2. 시스템은 ADL Code를 Parsing한다.    1. 어휘분석(Lexical analyzer)을 통해 각 문자를 Token으로 변환한다.    2. 구문분석(Syntax analyzing)을 통해 Token을 구조를 가진 구문 트리(Syntax Tree) 형태로 변경한다.    3. 의미분석(Semantic analyzer)을 통해 각 Syntax Tree의 관계를 분석해서 연결한 Parse Tree를 생성한다. 3. 시스템은 Parse Tree를 Graph 변환용 Language인 DOT과 같은 Graph Description Language Code로 변경한다. 4. 시스템은 Graph Description Language Code를 Graphviz와 같은 Graph Visualization Library에 Graph 변환 요청한다. 5. Graph Visualization Library는 시스템에 Graph Image를 반환한다. 6. 시스템은 ADL도구로 Graph image를 반환한다. |
| 추가 동작 |  |

|  |  |
| --- | --- |
| **UC\_02** | **Autocomplete** |
| 설명 | 입력중인 Code를 Prefix로 하는 Keyword들을 예측해서 반환한다. |
| 행위자 | ADL 도구 |
| 선행조건 |  |
| 후행조건 |  |
| 기본 동작 | 1. ADL 도구가 시스템으로 Autocomplete를 요청한다. 2. 시스템은 전달받은 문구로 시작하는 Keyword를 시스템이 정의한 Element, Relation, Attribute Type에서 검색한다. 3. 시스템은 검색된 모든 Keyword를 ADL 도구로 전달한다. |
| 추가 동작 |  |

|  |  |
| --- | --- |
| **UC\_03** | **Code Generate** |
| 설명 | 작성된 ADL code로 Android Skeleton Code를 생성한다. |
| 행위자 | ADL 도구 |
| 선행조건 |  |
| 후행조건 |  |
| 기본 동작 | 1. ADL 도구가 시스템으로 Code Generate를 요청한다. 2. 시스템은 ADL Code를 Parsing한다. 3. 시스템은 Parse Tree에서 View Type을 확인한다.    1. View Type의 name에 따라 package directory를 생성한다. 4. 시스템은 Parse Tree에서 Subsystem Type을 확인한다.    1. Subsystem의 Type(model, view, viewmodel, control, presenter)에 따라 package directory를 생성한다 5. 시스템은 Parse Tree에서 Component Type을 확인한다.    1. Component의 Type에 따라 Class를 생성한다.       1. Component Type이 Activity, Service, Broadcast Receiver, Content Provider인 경우 Android의 Activity, Service, BroadcastReceiver, ContentProvider Class를 extends한 Component name의 Class를 생성한다.    2. 이외의 Type은 Component Name으로 Class를 생성한다. 6. 시스템은 Parse Tree에서 Connector를 확인한다.    1. Connector의 ‘From’ Component에 대응하는 Class에 ‘To’ Component에 대응하는 Class를 Class member 변수로 추가한다. |
| 추가 동작 |  |

|  |  |  |
| --- | --- | --- |
| **NFR\_01** | **Reliability** | **시스템은 ADL Parsing 중 Error 발생시 빠르게 회복해야 한다.** |
| 설명 | 시스템이 ADL Code parsing중 문법 오류가 있는 경우, 해당 Line을 ADL 도구에 반환하고 다시 Translate 할 수 있는 상태로 빠르게 회복해야 한다. | |
| 환경 | 문법에 어긋난 ADL Code가 작성되어 있는 상황 | |
| 자극 | ADL 도구가 Translate 요청을 한다. | |
| 반응 | 시스템은 Error가 있는 Code Line을 ADL 도구에 반환하며 대시 Translate할 수 있는 상태로 회복한다. | |
| 측정 | [회복시간] = [Translate 요청을 받을 수 있는 상태로 회복된 시간] – [Error 감지된 시간] | |
| **제약** | 회복시간이 1000ms 이하여야 한다. | |

### ADL Specification

본 시스템은 ADL(Architecture Description Language)을 정의하며 Android Architecture 최적 설계를 위해 Component & Connector view와 Deployment View에 대한 Coding 방법을 명세한다.

Component & Connector view는 Android의 4대 Component인 Activity, Service, Broadcast Receiver, Content Provider와 Fragment를 명세한다. 또한 Android 대표적인 Style인 MVVM, MVC, MVP의 각 파트를 Subsystem으로 명세한다. 또한 일반적인 Application에서 사용되는 Database 등의 Component Type을 명세하고 각 Component들의 관계를 Connector로 연결한다.

Deployment View는 외부 System을 Component로 추가하고, Android의 System을 Process와 Thread 관점에서 각 Component들로 추가하고 각각의 관계를 Connector로 연결한다.

본 시스템에서 정의하는 Element Type은 아래와 같다.

|  |  |
| --- | --- |
| Element Type | Value |
| View | ComponentAndConnector  Deployment |
| Subsystem | Model  View  ViewModel  Controller  Presenter |
| Component | Activity  Service  Broadcast Receiver  Content Provider  Database  Component  Process  Thread  Server  Client  Cloud |

표 . Element Type

각 Element는 다양한 Attribute를 가질 수 있으며 Element를 고유하게 식별하는 Name, 해당 Element에 대한 설명을 위한 Description, Graph로 표시하는 본 시스템의 특성에 맞게 Color, Fill Color, Style, Shape을 설정한다.

|  |  |
| --- | --- |
| Attribute | Value |
| Name | Element의 구별 이름 |
| Description | Element에 대한 설명 |
| Color  FillColor | “0~255, 0~255, 0~255”  #FFFFFF  Red  Cyan  Blue  DarkBlue  LightBlue  Purple  Yellow  Lime  Magenta  Olive  Green  Maroon  Brown  Orange  Black  Gray  Silver  White |
| Style | solid  dashed  dotted  bold  invis  filled  diagonals  rounded |
| Shape | box  circle  ellipse  point |

표 . Element Attribute

본 시스템에서 정의하는 Relation Type은 아래와 같다.

|  |  |
| --- | --- |
| Relation Type | Value |
| Connector | Association  Dependency  Inheritance  Realization  Aggregation  Composition |

표 . Relation Type

Element와 마찬가지로 Attribute를 가지며, Name, Description과 더불어 Relation 특성상 연결의 시작점이 되는 from, 종착점이 되는 to, from과 to의 Element가 어느 방향으로 표시될지를 설정하는 Direction, 선의 색을 결정하는 Color가 있다.

|  |  |
| --- | --- |
| Attribute | Value |
| Name | Relation의 구별 이름 |
| Description | Relation에 대한 설명 |
| Color | “0~255, 0~255, 0~255”  #FFFFFF  Red  Cyan  Blue  DarkBlue  LightBlue  Purple  Yellow  Lime  Magenta  Olive  Green  Maroon  Brown  Orange  Black  Gray  Silver  White |
| Direction | up  down  left  right |

표 . Relation의 Attribute

## 비기능적 요구사항품질 속성

|  |  |  |
| --- | --- | --- |
| **QA\_01** | **Usability** | **ADL Code 작성의 편의성** |
| 설명 | ADL Code작성은 쉽고 간결하게 작성되야 한다. | |
| 환경 | 구조 설계자가 구조를 설계하려 한다. | |
| 자극 | ADL Code를 작성한다. | |
| 반응 | 작성된 ADL Code에 맞는 Graph를 반환한다. | |
| 측정 | [간결성] = [ADL Code 문자 수] / [변환된 Graph Description Language Code 문자 수]  작을수록 좋다. | |

|  |  |  |
| --- | --- | --- |
| **QA\_02** | **Performance** | **ADL Code의 Graph Description Language Code 변환 시간** |
| 설명 | ADL Code의 Graph Description Language로 변환이 빨라야 한다. | |
| 환경 | ADL Code 작성 상황 | |
| 자극 | ADL 도구가 Translate를 요청한다. | |
| 반응 | 시스템이 Graph Description Language로 변환한다. | |
| 측정 | [변환시간] = [Graph Description Language Code 완성 시간] – [Translate 요청 시간] | |

|  |  |  |
| --- | --- | --- |
| **QA\_03** | **Performance** | **Translate 요청 시 첫 Graph 반환 시간** |
| 설명 | ADL Code의 LOC에 관계없이 Translate 요청 시 Graph 반환이 빨라야 한다. | |
| 환경 | ADL Code 작성 상황 | |
| 자극 | ADL 도구가 Translate를 요청한다. | |
| 반응 | 시스템이 Graph를 반환한다. | |
| 측정 | [반환시간] = [처음으로 Graph를 반환한 시간] – [Translate 요청시간] | |

|  |  |  |
| --- | --- | --- |
| **QA\_04** | **Modifiability** | **Element Type 변경의 용이성** |
| 설명 | Element Type 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 환경 | Element Type 변경 요청 상황 | |
| 자극 | 고객이 개발팀에게 Element Type 변경을 요청한다. | |
| 반응 | 개발팀은 추가된 Element Type을 ADL상에서 사용할 수 있게 개발 / 검증 / 릴리즈 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC  작을수록 좋다. | |

|  |  |  |
| --- | --- | --- |
| **QA\_05** | **Modifiability** | **Graph Visualization Library 변경의 용이성** |
| 설명 | Graph Visualization Library 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 환경 | Graph Visualization Library 변경 요청 상황 | |
| 자극 | 고객이 개발팀에게 Graph Visualization Library 변경을 요청한다. | |
| 반응 | 개발팀은 변경된 Graph Visualization Library를 통해 Graph를 생성하도록 개발 / 검증 / 릴리즈 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC  작을수록 좋다. | |

|  |  |  |
| --- | --- | --- |
| **QA\_06** | **Performance** | **Autocomplete 완료 시간** |
| 설명 | Autocomplete 요청 시 빠르게 Prefix 문자에 해당하는 keyword list 반환해야 한다. | |
| 환경 | ADL Code 작성 상황 | |
| 자극 | ADL 도구가 Autocomplete 요청한다. | |
| 반응 | 요청된 Prefix로 시작하는 keyword list를 반환한다. | |
| 측정 | [반환시간] = [Keywork list 반환시간] – [Autocomplete 요청 시간] | |

|  |  |  |
| --- | --- | --- |
| **QA\_07** | **Extendibility** | **ADL Framework API의 확장성** |
| 설명 | ADL Framework API가 추가될 때 기존 Code에 큰 영향없이 쉽게 추가될 수 있어야 한다. | |
| 환경 | 시스템 Maintenance 상황 | |
| 자극 | 새로운 기능을 위한 ADL Framework API 추가 Requirement 발생 | |
| 반응 | 새로운 기능을 수행할 수 있는 ADL Framework API 추가 | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC  작을수록 좋다. | |

|  |  |  |
| --- | --- | --- |
| **QA\_08** | **Extendibility** | **Code Generate Language의 확장성** |
| 설명 | Code 생성시 지원하는 Programing Language가 쉽게 추가될 수 있어야 한다. | |
| 환경 | 시스템 Maintenance 상황 | |
| 자극 | 새로운 Programing Language 추가 Requirement 발생 | |
| 반응 | 새로운 Programing Language로 Code 생성할 수 있도록 시스템 수정 | |
| 측정 | [변경 Code 라인 수] = [새로운 Programing Language로 Code 생성 기능 추가 후 Code ]와 [기존 Code] 간에 다른 Code Line 수  작을수록 좋다. | |

# 시스템 구조

## 시스템 개요

ADL Framework은 하나의 정적 Library(AdlFramework.lib)와 다수의 동적 library로 발행된다. 아래 그림은 본 시스템의 artifact와 Process & Thread관점에서 기술된 Deployment View이다.



그림 . 전체 시스템 Deployment View

ADL 도구 개발자가 본 시스템을 활용해서 ADL Application을 개발하면, ADL Framework은 요청에 따라 동적 Library를 변경하며 동작하게 된다.

동작의 특성에 따라 3-tier로 구분되며, 각 Tier의 동작은 아래와 같다.

* Interface Tier는 ADL Application에 API를 제공하며 직접적으로 Interaction한다.
* Core Tier에서는 본 시스템의 기능인 Translate, Autocomplete, Code generation의 실질적인 동작을 수행한다.
* Extension Tier는 변경될 수 있는 Data나 Logic 및 외부Library를 동적으로 연결하며 제공한다.

본 시스템이 제공하는 기능인 Translate, Autocomplete, Code generation를 세가지 Task로 구성하고 모두 별도의 Thread로 동작한다. 또한 Translate의 어휘 분석, 구문 문석, 의미 분석 과정 및 Graph 전환 동작도 각각 별도 Thread로 동작하여 동시 작업성을 높였고 변경이 발생할 수 있는 부분을 별도 Library로 분리하여 수정 배포시의 변경 용이성을 높였다.

## 전체 시스템 구조



그림 . 전체 시스템 Component & Connector View

전체 시스템의 Component & Connector View는 위와 같다. 특정 작업을 수행하는 단위로 Component를 Grouping하였다. Interface Tier와 Core Tier와의 연결을 위해 Publish & Subscriber Architecture를 적용하였고, Core Tier 내부의 Translate 동작을 위해 Pipe & Filter Architecture를 적용하였다. 그리고 다양한 Extension Library와의 연결을 위해 Strategy, Bridge Pattern을 활용하였다.

Core tier와 Extension Tier를 연결하는 각각의 Library Interface는 아래의 목적으로 사용되며, 각각의 Concrete Library는 용도에 맞게 교체하며 사용 가능하다.

* IAppModel: 요구사항 2.1.2 ADL Specification에서 정의한 ADL의 Element, Relation, Attribute에 해당하는 Keyword list를 제공한다. Android 뿐 아니라 iOS, Windows의 특성에 맞는 Element들로 변경이 가능하다.
* IDrawer: Translate 동작 중 사용되는 Graph Description Language(이하 GDL)로 변환 기능을 제공한다. 각 Grape Visualization Library마다 지원하는 GDL이 다른데, Grape Visualization Library 변경 시 이에 지원이 가능한 GDL로 변경이 가능하다.
* IGraphDrawer: GDL을 Graph Visualization library를 통해 Graph로 변환하는 기능을 제공한다. 각각 다른 Graph Visualization library를 포함하고 있으며 이에 따라 조금씩 다른 look & feel을 보이는데, 이를 위해 Graph Visualization library를 변경할 수 있다.
* IClassProtoType: Code Generation시 제공하는Programming Language에 맞는 Package, Import, Class 선언, Member Field 선언 문구를 제공한다.

각 Component들의 자세한 동작은 다음 절에서 설명한다.

## 주요기능 동작 설명

### Program 구동 시 동작

Program 구동 시 각 Task에서 사용될 동적 Library에 대한 Component를 선택하게 되며 Default component는 config file을 통해 선정되나, ADL Framework API를 통해 동작 중에도 변경이 가능하다. 이렇게 생성된 Component Instance는 TaskManager에 의해 Task 생성시 Task에 주입된다.



그림 . Program 구동 시 동작

각 Task는 어떤 Library가 Instance화 됐는지에 관계없이 동일한 Interface로 동작하므로, 설정에 따른 다양한 동작이 가능하다.

### Translate 동작

ADL 도구로부터 Translate 요청과 함께 ADL Code를 전달받으면 Pipe & filter Architecture에 따라 여러 단계의 Parsing 작업후에 Graph를 반환하게 된다.



그림 . Translate Component & Connector View

본 시스템은 ADL을 자체적으로 정의했으며(참조. 후보구조 1. Element - Relation으로만 구성된 ADL) 이에 따라 해당 Language에 맞게 의미를 분석하고 Graph 변환 Language가 인식할 수 있는 Graph Description Language로 변환하는 작업이 필요하다. 즉 아래 순서대로의 변환 작업이 필요하며,

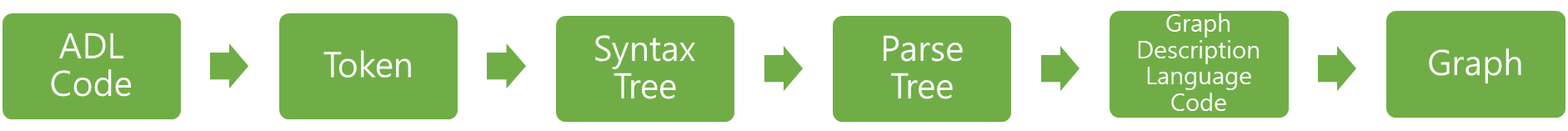


그림 . Translate Data 변환 과정

구체적인 Translate의 순서는 아래와 같다.



그림 . Translate Sequence Diagram

1. Translate 요청 시 TaskManager를 통해 TranslateTask를 생성한다.
2. 어휘분석(LexicalAnalyzer) Thread를 통해 ADL Code의 한 Line을 Tokenizing해서 Token Repository에 설정한다.
3. 설정된 Token을 Token Queue에 추가하고 어휘분석 Thread는 다음 Line Tokenizing을 다시 시작한다.
4. 구문분석(SyntaxAnalyzer) Thread는 Token Queue로부터 Token을 전달받고 Syntax Tree를 생성한다. 이때 성능 및 메모리 효율을 위해 Token Value를 따로 할당하지 않고 Token Repository를 참조하는 형태로 구성한다.
5. 분석된 Syntax Tree를 Syntax Queue에 추가하고 다시 Token Queue로부터 Token 수신을 대기한다.
6. 의미분석(SemanticAnalyzer) Thread는 Syntax Queue로부터 Syntax를 전달받고 Parse Tree를 생성한다.
7. 생성된 Parse Tree를 ScheduledQueue에 추가하고 다시 Syntax Queue로부터 Syntax Tree 수신을 대기한다.
8. ScheduledQueue는 Queue 전달 시간을 확인한다.
   1. 처음으로 추가된 ParseTree인 경우 즉시응답성을 위해 Drawer Thread로 전달하고
   2. 이후 추가되는 ParseTree는 너무 빈번한 Graph update를 방지하기 위해 일정 시간(500ms)동안 전달하지 않다가 해당 시간 경과 시 마지막으로 update된 ParseTree만 전달한다.
9. Drawer Thread는 ScheduledQueue로부터 ParseTree를 전달받고 Graph Description Language(이하 GDL) Code를 생성한다. Drawer별로 생성하는 GDL이 다르다.
10. Drawer Thread는 GDL 생성 이후에 GraphDrawer로 GDL을 전달한다.
11. GraphDrawer는 전달받은 GDL로 Graph를 생성하고 Translate Task로 반환한다.

각 단계에서 산출되는 Data 형태는 아래 Link된 부록 page의 Code와 그림을 참고하기 바란다.

* ADL Code - Code 2. MVVM Component & Connector View ADL Code
* Token - 그림 32. Token
* Syntax Tree - 그림 33. Syntax Tree
* Parse Tree - 그림 34. Parse Tree
* Graph Description Language Code - Code 3. MVVM Component & Connector View DOT code
* Graph - 그림 29. MVVM Component & Connector View

### Autocomplete 동작

Autocomplete 동작은 ADL도구가 특정 문구를 전달하면 이를 Prefix로 하는 Keyword list를 반환한다. 보다 빠른 성능을 위해 AppModel을 통한 Radix Tree(참조. 후보구조 18. Radix Tree로 Keyword 관리)를 생성하고 비교시에 사용한다. 또한 ADL 문맥에 따라 Element Type이나 Relation Type이 기입될 수 있는 상황과 Element의 Attribute Type이 기입될 수 있는 상황, 그리고 Relation의 Attribute Type이 기입될 수 있는 상황이 있으며 이를 구분하여 상황에 맞는 Keyword list를 반환한다.



그림 . Autocomplete Component & Connector View

문맥 분석을 위해서는 작성중인 Line의 ADL Code를 Tokenizing하고 이에 대한 구문 분석을 진행해야 하는데, Pipe & Filter 구조의 이점을 활용해서 Translate 동작의 SyntaxAnalyzer Filter를 AutocompleteSyntaxAnalyzer Filter로 교체하고 해당 Filter에서 Autocomplete에 맞는 구문 분석 방식을 적용한다.

구체적인 순서는 아래와 같다.



그림 . Autocomplete Sequence Diagram

1. AutocompleteTask가 LexicalAnalyzer(어휘분석)에 Tokenizing을 요청한다.
2. LexicalAnalyzer는 전달된 ADL Code를 Tokenizing해서 Token Repository에 설정한다.
3. 설정된 Token을 Token Queue에 추가한다.
4. Autocomplete를 위한 구문분석(AutocompleteSyntaxAnalyzer) Thread는 Token Queue로부터 Token을 전달받는다.
5. 문맥을 분석한다. (분석 방법 참조 - 후보구조 19. 문맥에 따른 Autocomplete 적용)
6. 분석된 문맥을 AutocompleteTask로 전달한다.
7. AutocompleteTask는 문맥에 따라 Element / Relation or Element Attribute or Relation Attribute에 맞는 keyword list를 Radix Tree에 요청한다.
8. Keyword list를 반환한다.

### Code Generate 동작

Code Generate 동작은 Parse Tree를 통해 package directory 및 Skeleton Class, Class의 Member Field를 생성한다.



그림 . Code Generate Component & Connector View

Code Generate동작은 다양한 Programming Language를 지원할 수 있는 구조를 가지며, AppModel을 통해 Element의 Class Name을 확인(참조. 표 9. Element Type Table)하고, ClassPrototype을 통해 package를 선언하는 방식, import 선언 방식, Class 정의 방식, Member field 정의 방식을 받아온다.

아래와 같은 순서로 진행이 되며, 실제 생성 예시는 ‘후보구조 23. Code Generation Library 적용’을 참조한다.



그림 . Code Generate Sequence Diagram

1. Code Generator로 ParseTree와 함께 Code 생성 요청한다.
2. AppModel로부터 Parse Tree의 root 부터 Node의 Type을 받아온다.
3. Type이 View Type인 경우 name에 해당하는 directory를 생성한다.
4. Type이 group Type인 경우 name에 해당하는 directory를 생성한다.
5. Type이 Component Type인 경우 AppModel로부터 Class Name을 받아온다.
6. ClassPrototype을 통해 package 선언 문구를 받아온다.
7. ClassPrototype을 통해 import 선언 문구를 받아온다.
8. ClassPrototype을 통해 Class 선언 문구를 받아온다.
9. Class를 생성한다.
10. Node에 Relation이 있는 경우 ClassPrototype을 통해 Member Field 선언문구를 받아온다.
11. 생성한 Class에 Member Field를 추가한다.

## 시스템 특성

### Extension tier의 변경 용이성

AppModel, Drawer, GraphDrawer, ClassPrototype 4가지 Interface를 정하고 이에 대한 Concrete module을 별도의 library로 생성하도록 구성함에 따라 다양한 조합의 동작이 가능하며, 추가 및 삭제 변경 시 ADL Framework 자체에 변경없이 Library 변경만으로도 적용이 가능한 구조이다.

### 성능 최적화

한 Line만으로도 Parse Tree를 생성할 수 있는 ADL을 정의함에 따라 Line별 병렬처리가 가능하고, Pipe & Filter Architecture를 적용하여 Parsing작업을 각각의 Thread로 동작하도록 구성해서 높은 성능을 발휘할 수 있도록 구성하였다.

### 3-tier 구조로 분리되어 독립적인 개발이 가능

전체 시스템이 Interface / Core / Extension Tier로 구분되어 있고, Core Tier로만의 단방향 Dependency가 존재한다. 이에 따라 각 Tier별 독립적인 개발이 가능한 구조이다.

# 모듈 사양

본 시스템의 전체 모듈 구조는 아래와 같다.



그림 . 전체 시스템 Module View

총 3개의 Layer로 나뉘어지며 ADL 도구와 Interaction하는 Interface layer와 시스템의 주요 logic을 수행하는 Core Layer, 변경될 수 있는 Data나 Logic 및 외부 library를 동적으로 연결하는 Extension Layer가 있다.

## 모듈 상세

### Interface Layer – Core Layer

Interface Layer와 Core Layer 경계의 모듈 상세 구조는 아래와 같다.



그림 . Interface Layer – Core Layer Class Diagram

Interface Layer에는 ADL 도구가 호출할 수 있는 Module인 AdlManager가 존재한다. ADL 도구가 AdlListener의 Concrete Module을 생성해서 AdlManager에게 등록해두면 translate() 및 autocomplete() codegenerate()에 대한 결과를 비동기적으로 받을 수 있다. ADL 도구는 결과를 확인하고 Graph를 표시하거나 Autocomplete Guide를 제공하거나, Code Generate 완료 Dialog를 생성하는 등의 작업을 하게 된다.

TaskManager는 Singleton 객체이며, AdlManager는 ITaskManager Interface를 통해 TaskManager API를 호출한다. AdlManager와 TaskManager는 Acyclic Dependencies Principle에 따라 Observer pattern을 통해 서로 Communication할 수 있도록 구성하였다. 이를 위해서 AdlManager는 ITaskObserver의 Concrete 객체를 TaskManager에 등록한다.

### Core Layer – Extension Layer

Core Layer와 Extension Layer 경계의 모듈 상세 구조는 아래와 같다.



그림 . Core – Extension Layer Class Diagram

Extension Layer의 Module들은 Core Layer의 4가지 interface IAppModel, IDrawer, IGraphDrawer, IClassPrototype중 하나를 상속받아서 구현한 Concrete Module이다. TaskManager는 각각의 Concrete Module을 가지고 있고 Task 생성시 Task에 필요한 Module을 주입한다.

* IAppModel

현재 ADL에서 Keyword로 인식할 수 있는 Element와 Relation, Attribute값을 가지고 있으며 각각에 대한 List를 View, SubSystem, Component, Relation, Element Attribute, Relation Attribute Type에 맞게 반환할 수 있다. 이는 Translate시 Syntax 분석에 쓰이며, 또한 Autocomplete시에 Prefix에 따른 추천 Keyword List생성에 쓰인다. 또한 Code Generation을 위해 현재 Element가 어떤 Type인지를 확인할 수 있는 API와 현재 Component에 해당하는 실제 Class name을 반환하는 API를 제공한다. Strategy Pattern을 적용해서 상황에 따라 변경하며 사용이 가능하다.

* IDrawer & IGraphDrawer

IDrawer와 IGraphDrawer는 서로 조합에 연관이 있는 Module로 Bridge Patten으로 적용되었다.



그림 . IDrawer – IGraphDrawer Bridge Pattern

IDrawer는 Parse tree를 바탕으로 Graph Description Language Code로 변환하는 역할을 하는 Module이며 어떤 Language로 변환할지에 따라 Concrete Module이 변경된다. Parse Tree는 각 Element의 Attribute 및 계층 관계가 모두 표현되어 있어서, Parse Tree를 통한 Graph Description Language 변환은 해당 Language에 대응하는 단순 Mapping으로 변환이 가능하다. Graph Description Language는 Graph 변환에 사용되는 Graph Visualization Library와 밀접한 관계가 있는데, 각 Graph Visualization Library마다 인식할 수 있는 Graph Description Language가 한정적이기 때문이다. 이에 따라 Bridge Pattern을 적용해서 IDrawer 생성시 IGraphDrawer를 주입시키며, IDrawer의 Graph Description Language변환 작업이 모두 완료되면 IGraphDrawer로 graph 변환을 요청한다.

IGraphDrawer는 Graph 전환 동작을 하는 외부 Library를 포함하는 Module로써, IDrawer로부터 전달받은 Graph Description Language Code를 각 Concrete Module이 가지고 있는 Graph Visualization Library에 Graph 전환 요청을 한다. 변경완료시 Graph Image를 IDrawer로부터 전달받은 Listener에게 반환한다.

* IClassProtoType

IClassProtoType은 CodeGeneration시에 각 Programming Language별로 차이가 발생할 수 있는 구문을 제공하는 Module이다. Strategy Pattern으로 필요에 따라 Concrete Module을 변경하며 동작할 수 있다. Class를 생성할 때, package name을 전달하면 package를 선언하는 문구를 반환하고, import를 하려는 full address Class name을 전달하면 이를 import하는 문구를 반환하고, 특정 Class를 상속받아서 특정 Name으로 Class를 생성하려고 할 때 super Class Name과 Class Name을 전달하면 Class를 생성하는 문구를 반환하고, 특정 Type으로 특정 Name의 Class member field를 생성하려고 할 때 이 Class Type과 Name을 전달하면 Class member field를 생성하는 문구를 반환한다. 좀 더 자세한 설명은 ‘후보구조 24. Language별 Class Prototype변경’을 참조한다.

### Translate Task

Translate Task 관련 모듈 상세 구조는 아래와 같다.



그림 . Translate Task Class Diagram

StateManager는 현재 Task가 실행 중인지 아닌지를 판단하는 모듈로 TaskManager에 의해 활용된다. Task Interface를 상속받은 TranslateTask는 생성시에 주입받은 IDrawer와 IAppModel을 Member Field로 가지고 있으며, Pipe & Filter Architecture의 각 Filter들(LexicalAnalyzer, SyntaxAnalyzer, SemanticAnalyzer)도 포함한다. 각 Filter들은 생성한 Data를 담을 Queue들을 가지고 있으며, 해당 Queue를 통해 동작할 다음 순번의 Queue를 등록할 수 있는 API를 제공한다. LexicalAnalyzer는 ADL Code를 tokenizing하는 모듈이며 TokenRepository는 Token들을 매번 생성하지 않도록 일정 Memory를 생성해둔 저장소이다. SyntaxAnalyzer는 token을 분석해서 SyntaxTree를 생성하는 모듈로 SyntaxTree는 Token의 의미에 맞게 구조화된 Data이다. SemanticAnalyzer는 SyntaxTree들을 분석하고 각각의 관계를 정리해서 ParseTree를 생성하는 모듈이며 Parse Tree는 각 SyntaxTree들의 관계가 표현된 Data이다.

### Autocomplete Task

Autocomplete Task관련 모듈 상세구조는 아래와 같다.



그림 . Autocomplete Task Class Diagram

Autocomplete Task는 Translate Task의 Pipe & Filter 구조에서 SyntaxAnalyzer Filter를 AutocompleteSyntaxAnalyzer Filter로 교체해서 사용한다. AutocompleteSyntaxAnalyzer Module은 Token의 문맥을 확인하는 Method를 제공하며, 완료 후 등록된 Listener에게 event를 전달한다. RadixTree Module은 AppModel을 통해 생성한 Radix Tree로 각 Type별 keyword list를 반환해주는 search 기능을 제공한다.

### CodeGenerate Task

CodeGenerate Task 관련 모듈 상세구조는 아래와 같다.



그림 . CodeGenerate Task Class Diagram

CodeGenerateTask는 Parse Tree를 기준으로 Package Directory 및 Class를 생성한다. Parse Tree는 Root node와 다음 Node를 가져올 수 있는 메서드를 제공하고 Code Generator가 이를 통해 IAppModel로부터 현재 Node의 Type을 확인한다. FileManager는 package 이름에 따라 directory를 생성할 수 있는 createPackage API를 제공하고, class를 생성하는 API, Field를 추가하는 API를 제공한다. 어떠한 문구로 생성이 되는지는 Code Generator가 IClassPrototype을 통해 Programming Language 특성에 따른 문구를 확인후 전달한다.

## Work Assignment

본 시스템의 개발팀 구성은 다음과 같다. 3개의 Layer로 구분된 구성대로 개발 그룹이 나뉘어지고 Core Layer는 또 Task별로 개발 파트가 나뉘어진다.



부록

[A. 도메인 모델 43](#_Toc69756673)

[B. 품질 시나리오 47](#_Toc69756674)

[C. 품질 시나리오 분석 49](#_Toc69756675)

[D. 후보 구조 51](#_Toc69756676)

[D1. QA\_01 ADL Code 작성의 편의성 51](#_Toc69756677)

[D1.1. 후보구조 1. Element - Relation으로만 구성된 ADL 51](#_Toc69756678)

[D1.2. 후보구조 2. 기존 Markup language의 문법 사용 54](#_Toc69756679)

[D1.3. 후보구조 3. Programming Language형태의 ADL 명세 55](#_Toc69756680)

[D1.4. 후보구조 4. 확장 가능한 Component Type 57](#_Toc69756681)

[D2. QA\_02 ADL Code의 Graph Description Language Code 변환 시간 58](#_Toc69756682)

[D2.1. 후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계 58](#_Toc69756683)

[D2.2. 후보구조 6. Update를 위해 Token 및 Parse Tree 유지 63](#_Toc69756684)

[D2.3. 후보구조 7. Token, Tree 구성을 위한 Resource 선 할당 64](#_Toc69756685)

[D2.4. 후보구조 8. Tree 구성을 위한 Resource 효율화 64](#_Toc69756686)

[D2.5. 후보구조 9. Dot language Parser library를 사용 64](#_Toc69756687)

[D3. QA\_03 Translate 요청 시 첫 Graph 반환 시간 65](#_Toc69756688)

[D3.1. 후보구조 10. 연속적으로 Graph를 갱신 65](#_Toc69756689)

[D3.2. 후보구조 11. 일정한 주기로 Graph를 갱신 66](#_Toc69756690)

[D3.3. 후보구조 12. Translate 요청전에 Background parsing 67](#_Toc69756691)

[D4. QA\_04 Element Type 변경 용이성 67](#_Toc69756692)

[D4.1. 후보구조 13. 별도의 AppModel library로 지연 바인딩 67](#_Toc69756693)

[D4.2. 후보구조 14. Element Type에 Decorate Pattern을 적용 69](#_Toc69756694)

[D4.3. 후보구조 15. Element Type에 Strategy Pattern을 적용 69](#_Toc69756695)

[D5. QA\_05 Graph Visualization Library의 변경 용이성 70](#_Toc69756696)

[D5.1. 후보구조 16. 별도의 GraphDrawer Library로 지연 바인딩 71](#_Toc69756697)

[D5.2. 후보구조 17. GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용 73](#_Toc69756698)

[D6. QA\_06 Autocomplete 완료시간 73](#_Toc69756699)

[D6.1. 후보구조 18. Radix Tree로 Keyword 관리 74](#_Toc69756700)

[D6.2. 후보구조 19. 문맥에 따른 Autocomplete 적용 74](#_Toc69756701)

[D6.3. 후보구조 20. ADL 도구에 Keyword Array 전달 75](#_Toc69756702)

[D7. QA\_07 ADL Framework API의 확장성 76](#_Toc69756703)

[D7.1. 후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용 76](#_Toc69756704)

[D7.2. 후보구조 22. Task 기반의 Event Bus 적용 77](#_Toc69756705)

[D8. QA\_08 Code 생성 Language 확장성 78](#_Toc69756706)

[D8.1. 후보구조 23. Code Generation Library 적용 78](#_Toc69756707)

[D8.2. 후보구조 24. Language별 Class Prototype변경 81](#_Toc69756708)

[D9. NFR\_01. ADL Parsing 중 Error 발생시 회복 시간 83](#_Toc69756709)

[D9.1. 후보구조 25. ADL 분석 단계에서 Error 발생시 Thread 종료 84](#_Toc69756710)

[D9.2. 후보구조 26. Translate 동작의 Multitasking 84](#_Toc69756711)

[E. 후보 구조 평가 86](#_Toc69756712)

[E1. QA\_01 ADL Code 작성의 편의성에 관한 후보구조 평가 86](#_Toc69756713)

[E2. QA\_02 ADL Code의 Graph Description Language 변환성능에 관련한 후보구조 평가 88](#_Toc69756714)

[E3. QA\_03 Translate 요청 시 첫 Graph 반환 성능 관련 후보구조 평가 90](#_Toc69756715)

[E4. QA\_04 Element Type 변경 용이성 관련 후보구조 평가 91](#_Toc69756716)

[E5. QA\_05 Graph Visualization Library 변경 용이성 관련 후보구조 평가 92](#_Toc69756717)

[E6. QA\_06 Autocomplete 완료 성능 관련 후보구조 평가 93](#_Toc69756718)

[E7. QA\_07 ADL Framework API의 확장성 관련 후보구조 평가 94](#_Toc69756719)

[E8. QA\_08 Code 생성 Language 확장성 관련 후보구조 평가 95](#_Toc69756720)

[E9. NFR\_01 ADL Parsing중 Error 발생시 회복 시간 후보 구조 검토 95](#_Toc69756721)

[E10. 후보 구조 평가 결과 96](#_Toc69756722)

[F. 최종 구조 설계 99](#_Toc69756723)

[F1. 시스템의 변경용이성 측면 99](#_Toc69756724)

[F1.1. Extension Layer – Core Layer 100](#_Toc69756725)

[F1.2. Interface Layer – Core Layer 101](#_Toc69756726)

[F2. 시스템의 성능 측면 102](#_Toc69756727)

[F2.1. Translate 성능 104](#_Toc69756728)

[F2.2. Autocomplete 성능 105](#_Toc69756729)

[F3. 시스템의 신뢰성 측면 106](#_Toc69756730)

[F4. 시스템의 Risk & Management 107](#_Toc69756731)

[F4.1. 사용성에 의한 성능 Risk 107](#_Toc69756732)

[F4.2. 성능에 의한 변경용이성 Risk 108](#_Toc69756733)

[G. 최종 구조 평가(ATAM) 109](#_Toc69756734)

1. 도메인 모델

본 과제의 도메인 모델은 아래 그림과 같다.

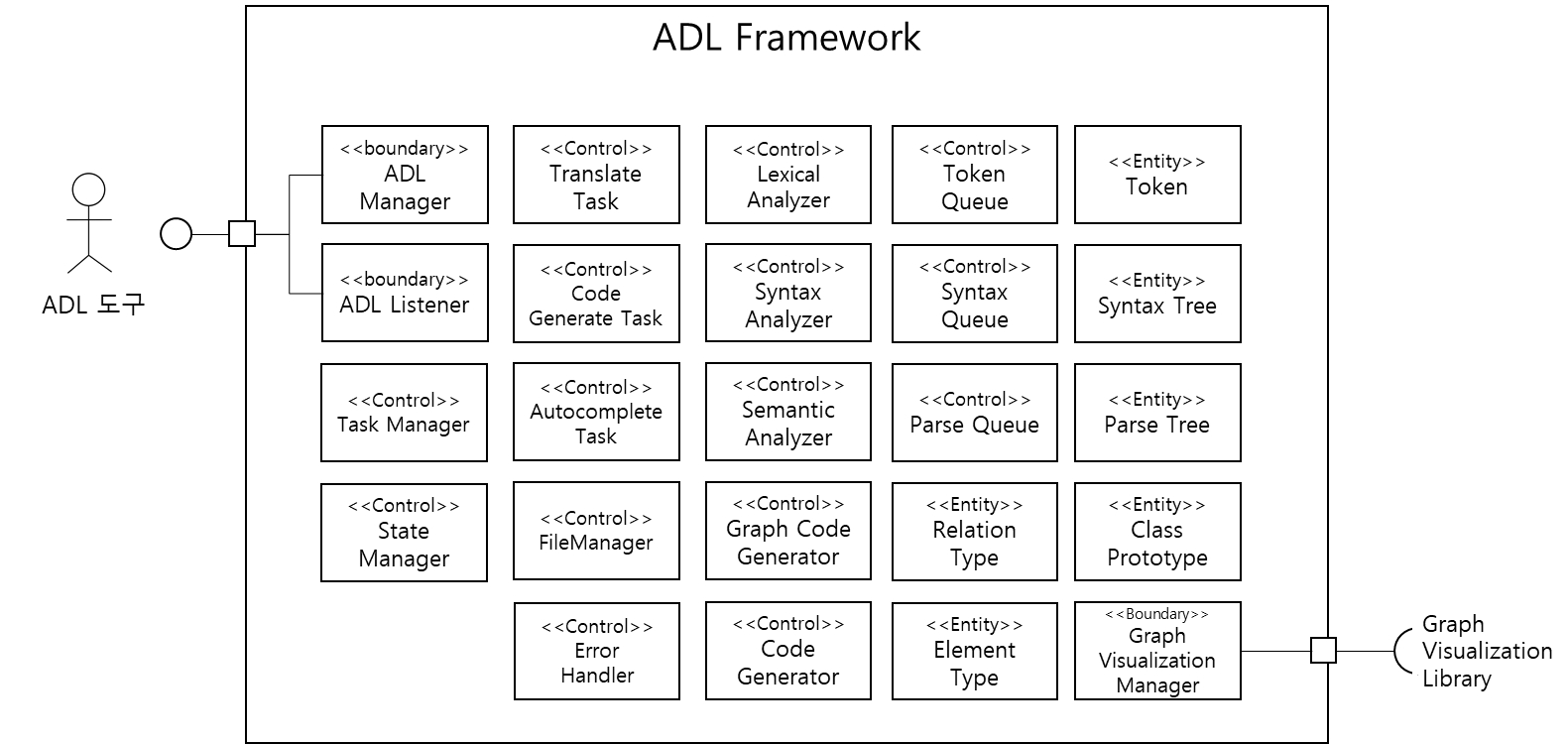


그림 . Domain Model

각 Use Case에 대한 세부적인 Sequence Diagram은 아래와 같다.



그림 . UC\_01 Translate



그림 . UC\_02 Autocomplete



그림 . UC\_03 Code Generate

1. 품질 시나리오

본 시스템의 품질 시나리오 들은 아래와 같다.

|  |  |  |
| --- | --- | --- |
| **QS\_01** | **Usability** | **ADL Code 작성의 간결성** |
| 설명 | ADL Code작성이 쉽고 직관적이며 간결하게 작성되야 한다. | |
| 측정 | [간결성] = [ADL Code 문자 수] / [변환된 Graph Description Language Code 문자 수] | |
| **QS\_02** | **Performance** | **Autocomplete 완료시간** |
| 설명 | Autocomplete 요청 시 문자열 반환하는 시간이 빨라야 한다. | |
| 측정 | [반환시간] = [문자열 반환 시간] – [Autocomplete 요청시간] | |
| **QS\_03** | **Performance** | **ADL Code의 Graph Description Language Code 변환 시간** |
| 설명 | ADL Code의 Graph Description language로 변환이 빨라야 한다. | |
| 측정 | [변환시간] = [Graph Description Code 완성 시간] – [Translate 요청 시간] | |
| **QS\_04** | **Performance** | **Translate 요청 시 첫 Graph 반환 시간** |
| 설명 | ADL Code의 LOC에 관계없이 Translate 요청 시 Graph 반환이 빨라야 한다. | |
| 측정 | [반환시간] = [처음으로 Graph를 반환한 시간] – [Translate 요청시간] | |
| **QS\_05** | **Performance** | **Code Generate시 Code 생성 시간** |
| 설명 | Code Generate시 Code 생성이 빨라야 한다. | |
| 측정 | [코드 생성시간] = [생성 완료시간] – [Code Generate 요청시간] | |
| **QS\_06** | **Compatibility** | **기존 ADL Tool과의 호환성** |
| 설명 | Acme Studio등과 같은 기존 ADL tool에서 인식할 수 있도록 file export할 수 있어야 하고 해당 Tool에서 작성된 file도 import 할 수 있어야 한다. | |
| 측정 | [Import / Export 성공률] = [Import / Export 성공 Tool수] / [Total ADL Tool수] | |
| **QS\_07** | **Modifiability** | **Element Type 변경 용이성** |
| 설명 | Element Type 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC | |
| **QS\_08** | **Modifiability** | **Graph Visualization Library 변경 용이성** |
| 설명 | Graph Visualization Library 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC | |
| **QS\_09** | **Interoperability** | **ADL 공유 작업 동시성** |
| 설명 | 다수의 구조 설계자가 ADL Code를 공유하며 동시작업이 지연없이 동작해야 한다. | |
| 측정 | [지연율] = [다른 구조 설계자가 변경된 ADL Code를 확인하는 시간] – [구조 설계자가 ADL Code를 변경하는 시간] | |
| **QS\_10** | **Reliability** | **ADL Parsing 중 Error 발생시 회복 시간** |
| 설명 | ADL Code parsing중 문법 오류가 있는 경우, 해당 Line을 ADL 도구에 반환하고 다시 Translate 할 수 있는 상태로 빠르게 회복해야 한다. | |
| 측정 | [회복시간] = [Translate 요청을 받을 수 있는 상태로 회복된 시간] – [Error 감지된 시간] | |
| **QS\_11** | **Extendibility** | **Code Generate Language 확장성** |
| 설명 | Code Generate시 지원하는 Programing Language가 쉽게 추가될 수 있어야 한다. | |
| 측정 | [변경 Code 라인 수] = [새로운 Programing Language로 Code 생성 기능 추가 후 Code ]와 [기존 Code] 간에 다른 Code Line 수 | |
| **QS\_12** | **Extendibility** | **ADL Framework API의 확장성** |
| 설명 | ADL Framework에서 제공하는 API의 추가가 기존 모듈에 큰 영향없이 추가될 수 있어야 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC | |

1. 품질 시나리오 분석

품질 시나리오의 중요도와 복잡도에 따라 최종 선택된 NFR(Non Functional Requirements), QA(Quality Attribute) 그리고 QS(Quality Scenario)의 매핑은 아래 표와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 구분 | QS\_ID | 항목 | 중요도 | 복잡도 | 속성 |
| Usability | QS\_01 | ADL Code 작성의 편의성 | H | H | QA\_01 |
| Performance | QS\_02 | Autocomplete 완료시간 | H | M | QA\_06 |
| QS\_03 | ADL Code의 Graph Description Language Code 변환 시간 | H | H | QA\_02 |
| QS\_04 | Translate 요청 시 첫 Graph 반환 시간 | H | H | QA\_03 |
| QS\_05 | Code Generate시 Code 생성 시간 | L | H |  |
| Compatibility | QS\_06 | 기존 ADL Tool과의 호환성 | L | H |  |
| Modifiability | QS\_07 | Element Type 변경 용이성 | H | H | QA\_04 |
| QS\_08 | Graph Visualization Library 변경 용이성 | H | H | QA\_05 |
| Interoperability | OS\_09 | ADL 공유 작업 동시성 | L | H |  |
| Reliability | QS\_10 | ADL Parsing 중 Error 발생시 회복 시간 | H | M | NFR\_01 |
| Extendibility | QS\_11 | Code Generate Language 확장성 | H | M | QA\_08 |
| QS\_12 | ADL Framework API의 확장성 | H | M | QA\_07 |

본 시스템은 ADL Language를 정의하고 이를 사용해서 즉각적으로 손쉽게 Android Application Architecture를 그릴 수 있도록 하는데 그 목적이 있다. 이에 따라 **QS\_01. ADL Code 작성의 편의성**은 실제 Coding을 하는 구조 설계자가 ADL을 익히는 시간을 최소화해서 사용성을 높일 수 있는 부분으로 높은 중요도를 가지며 **QA\_01**으로 선정한다.

그리고 작성한 ADL Code를 Graph로 변환하는 성능도 높은 중요도를 가지며, Graph 변환의 중간단계인 **QS\_03. ADL Code의 Graph Description Language Code 변환 시간**을 **QA\_02**로 선정한다. 더불어 Translate 요청 시 Grape 반환시간은 전체 코드의 Graph Description Language 변환 이전에 Translate 요청에 즉각적으로 Graph를 반환하여 즉시 응답성을 확보할 필요가 있다. 이에 따라 **QS\_04 Translate 요청 시 첫 Graph 반환 시간**을 **QA\_03**로 선정한다.

**QS\_05 Code Generate시 Code 생성 시간은** Code Generate 동작 자체가 사용자가 ADL Code 작성 완료후에 실행하는 만큼 자주 사용하는 기능이 아닐 것으로 판단되며 생성 시간 보다는 기능적으로 오류 없이 동작하는 것 만으로도 충분한 기능으로 판단되어 품질 속성에서 제외한다.

**QS\_06 기존 ADL Tool과의 호환성**은 다른 Tool이 본 시스템의 ADL에 맞게 추가 개발을 하거나, 본 시스템에서 기존 ADL에 맞게 변환하는 작업이 필요하며, 이를 위해서는 기존의 모든 ADL을 분석하고 변환해야 한다. 이는 시스템 개발 전체 M/M에 상당부분을 차지할 만한 난이도이나 이를 통해 얻을 수 있는 이점이 미미하여 품질속성에서 제외한다.

Element Type은 요청에 따라 빈번히 변경이 이루어질 부분이어서 **QS\_07. Element Type 변경 용이성**은 본 시스템의 중요한 품질 속성이며 **QA\_04**로 선정한다. 또한 **QS\_08. Graph Visualization Library 변경 용이성**도 요구사항 변경에 따라 변경 가능성이 높은 부분으로 품질 속성 **QA\_05**로 선정한다.

하지만 **QS\_09 ADL 공유 작업 동시성**은 Architecture 설계과정이 일반적으로 공동 작업으로 진행되지 않는 점과 동시성 확보를 위한 비용이 큰 점을 이유로 품질속성에서 제외한다.

**QS\_10. ADL Parsing중 Error 발생시 회복 시간**은 Language 특성상 자주 발생할 수 있는 부분이며, 이에 따른 회복 시간이 일정시간 이하로 유지되야 해서 **NFR\_01**로 선정한다.

또한 **QS\_02. Autocomplete 완료시간**도 Language 특성상 자주 사용될 수 있는 동작으로 성능이 높을수록 작성의 편의성을 높일 수 있기 때문에 품질속성 **QA\_06**으로 선정한다.

**QS\_10. ADL Framework API**의 확장성은 시스템 자체의 새로운 기능 추가 시 보다 수월하게 이를 적용하기 위한 품질 속성으로 **QA\_07**로 선정한다.

**QS\_11. Code 생성 Language 확장성**은 Android Application이 Java가 아닌 Kotlin등의 프로그래밍언어로 확대되고 있는 상황인만큼 **QA\_08** 선정해서 추후 확장성을 확보한다.

1. 후보 구조
   1. QA\_01 ADL Code 작성의 편의성

사용자가 ADL Code 작성이 복잡하다면, 본 시스템을 사용하지 않고 다른 ADL을 사용하거나 PPT, 종이에 Architecture를 그릴 것이다. 즉 ADL Code 작성의 편의성은 본 시스템의 가장 중요한 품질 속성으로 이에 대한 후보 구조를 각각 검토한다.

* + 1. 후보구조 . Element - Relation으로만 구성된 ADL

Component & Connector view나 Deployment view나 모두 큰 범주로는 Element들 간의 Relation으로 볼 수 있다. 이에 따라 ADL 자체를 Element와 Relation 두 가지 형태로만 정의하고 각각의 Type과 Attribute를 아래와 같이 기술한다.

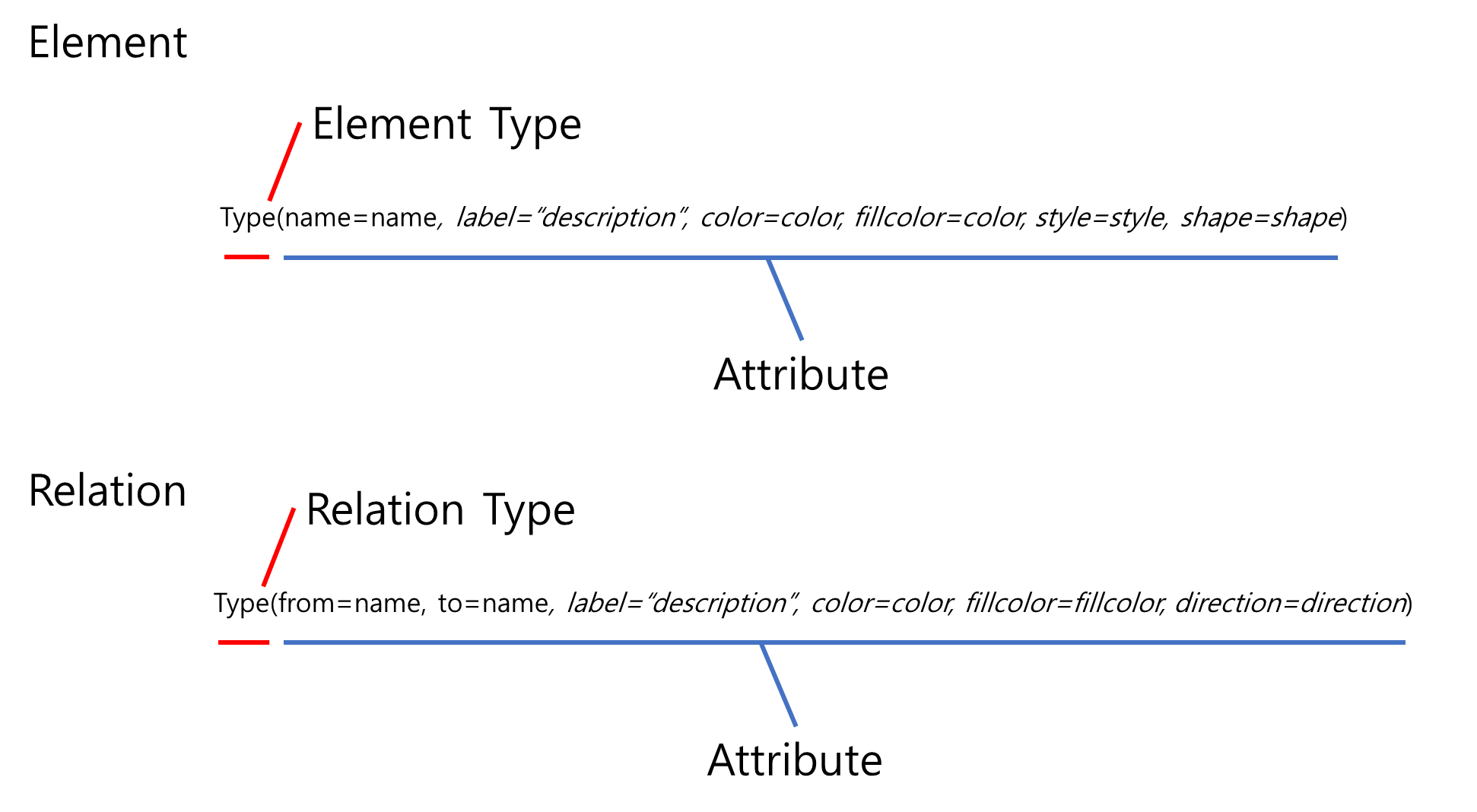


그림 . Element와 Relation 기술 방법

각 Element들은 Group으로 표현될 수 있으며, 들여쓰기(4 / 8 space or tab)를 통해 Group을 설정한다.

예를 들어 아래와 같이 View Element 아래에 Fragment와 Activity Element를 들여쓰기로 선언 시 Fragment와 Activity는 View의 Child로 표현된다.

|  |
| --- |
| **View**(name=view)  **Fragment**(name=myFragment)  **Activity**(name=myActivity) |

Code . Child 선언하는 ADL Code

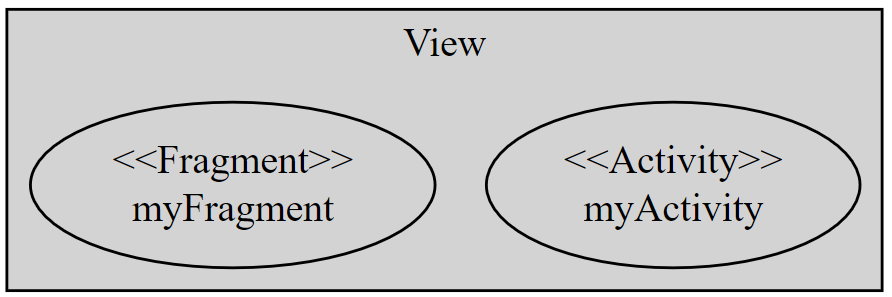


그림 . View의 Child로 선언된 Fragment와 Activity의 Graph

본 시스템은 Android Application Architecture를 보다 손쉽게 설계하는데 그 목적이 있는 만큼, Android Application의 기본적인 Architecture인 MVVM과 MVC, MVP Style을 Element Type으로 선언할 수 있다. 이에 따라 Component & connector View로 아래와 같이 기본적인 MVVM구조를 작성할 수 있다.

|  |
| --- |
| ComponentAndConnector(name=deploymentView, description=”Component & Connector View”)  View(name=view, fillcolor=RED)  Fragment(name=myFragment)  Activity(name=myActivity)  Association(from=myFragment, to=myActivity)  ViewModel(name=viewModel, fillcolor=BLUE)  Component(name=lifeCycleObserver)  Component(name=liveData)  Model(name=model, fillcolor=GREEN)  Database(name=myDB)  Service(name=myService)  Cloud(name=myCloud, fillcolor=WHITE)  Association(from=myActivity, to=liveData)  Association(from=liveData, to=myService)  Aggregation(from=myCloud, to=myService) |

Code . MVVM Component & Connector View ADL Code

본 시스템은 Graphviz와 같은 Graph Visualization Library를 사용해서 Graph를 생성한다. 이에 따라 각 Library를 위한 Language 변환이 필요하며, 아래 Code는 Graphviz에서 사용하는 DOT Graph Description Language로 변환한 Code이다.

|  |
| --- |
| digraph G {  label = "Component & Connector View"  subgraph cluster0 {  style=filled;  myActivity[label="<<Activity>>\nmyActivity"]  myFragment[label="<<Fragment>>\nmyFragment"]  label = "View";  fillcolor = RED  }  subgraph cluster1 {  style=filled;  liveData[label="<<Component>>\nliveData"]  lifeCycleObserver[label="<<Component>>\nlifeCycleObserver"]  label = "ViewModel";  fillcolor = green  }    subgraph cluster2 {  style=filled;  label = "Model";  myService[label="<<Service>>\nmyService"]  DB[label="<<DataBase>>\nmyDB"]  color=lightblue  }  myCloud[label="<<Cloud>>\nmyCloud"]    edge[ weight = 5 ]  myActivity -> liveData  liveData -> myService  myCloud -> myService[arrowhead="odiamond"]  } |

Code . MVVM Component & Connector View DOT code

변경된 DOT Code를 Graphviz에서 Graph로 변환한 결과는 아래와 같다.

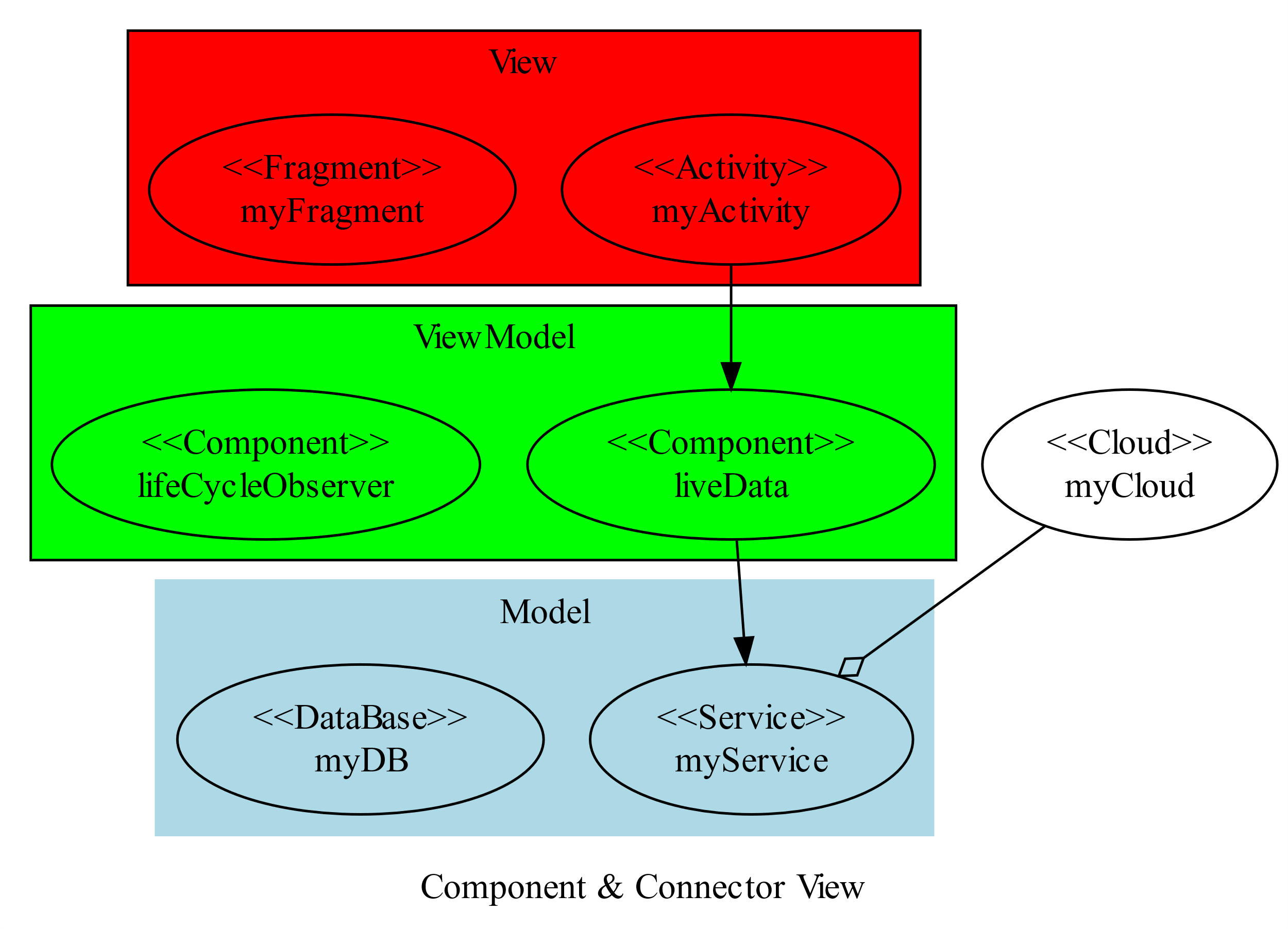


그림 . MVVM Component & Connector View

* + 1. 후보구조 . 기존 Markup language의 문법 사용

XML이나 HTML, Json과 같은 널리 사용되는 Markup Language의 문법을 그대로 적용해서 별도로 ADL을 정의하는 비용을 절감하고, 해당 language가 익숙한 개발자에겐 ADL 학습에 대한 부담을 줄일 수 있다. 이후 Parsing 단계에서도 기존 language를 사용하면 기 존재하는 다양한 Parsing Library들을 그대로 사용할 수 있어서 개발비용을 절감할 수 있는 장점도 있다. 하지만 사람이 이해하기 쉬운 관점이 아닌 기계가 이해하기 쉽게 만든 Language인만큼 Code 작성이 직관적이지 않을 수 있다.

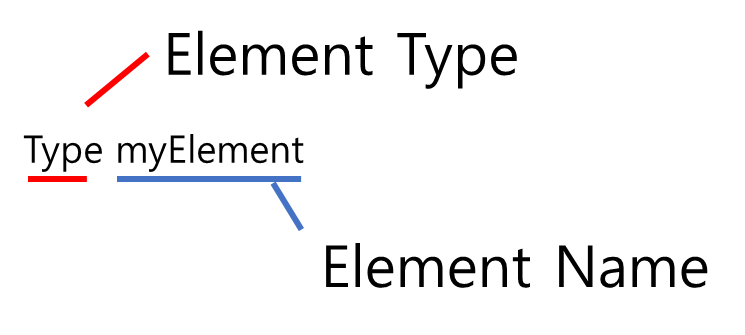
후보구조 1에서 예시로 사용된 MVVM Code를 XML로 표현하면 아래와 같다.

|  |
| --- |
| <view type=ComponentAndConnector name=componentView description="Component & Connector View">      <subsystem type=View name=view fillcolor=RED>          <component type=Fragment name=myFragment/>          <component type=Activity name=myActivity/>          <connector type=Association name=myFragment/>      </subsystem>      <subsystem type=ViewModel name=viewModel fillcolor=BLUE>          <component type=Component name=lifeCycleObserver/>          <component type=Component name=liveData/>      </subsystem>      <subsystem type=Model name=model fillcolor=GREEN>          <component type=Database name=myDB/>          <component type=Service name=myService/>      </subsystem>      <component type=Cloud name=myCloud fillcolor=WHITE/>      <connector type=Association from=myActivity to=liveData/>      <connector type=Association from=liveData to=myService/>      <connector type=Aggregation from=myCloud to=myService/>  </view> |

Code . XML for ADL

* + 1. 후보구조 . Programming Language형태의 ADL 명세

객체를 선언하고, 해당 객체의 Method를 호출하는 형태, Programming Language 형태로 ADL을 작성하면, Programming이 익숙한 구조 설계자가 좀 더 쉽게 ADL을 익힐 수 있을 것이다. Element 선언은 아래와 기술하며 모든 Attribute 및 Relation, Grouping은 Method로 설정한다.



Element에 지원하는 Method는 아래와 같다.

|  |  |
| --- | --- |
| Method | Description |
| setDescription(description) | Element의 description을 설정한다. |
| setColor(color) | Element 선의 색을 설정한다. |
| setFillColor(color) | Element의 채우기 색을 설정한다. |
| setStyle(style) | Element의 Style을 설정한다. |
| setShape(shape) | Element의 Shape를 설정한다. |
| associate(otherElement) | Element의 Association 대상 Element를 설정한다. |
| dependent(otherElement) | Element의 Dependenty 대상 Element를 설정한다. |
| inherit(otherElement) | Element의 Inheritance 대상 Element를 설정한다. |
| realize(otherElement) | Element의 Realization 대상 Element를 설정한다. |
| aggregate(otherElement) | Element의 Aggregation 대상 Element를 설정한다. |
| compose(otherElement) | Element의 Composition 대상 Element를 설정한다. |
| addChild(otherElement) | Element에 특정 Element를 추가해 Grouping한다. |

이를 통해 후보구조 1에서 예시로 사용한 MVVM Code를 작성하면 아래와 같다.

|  |
| --- |
| DeploymentView deploymentView      deploymentView.setDescription("Deployment View")      View view      Fragment myFragment      Activity myActivity      view.setColor(RED)      view.addChild(myFragment)      view.addChild(myActivity)      fragment.associate(myActivity)      ViewModel viewModel      Component lifeCycleObserver      Component liveData      viewModel.setColor(BLUE)      viewModel.addChild(lifeCycleObserver)      viewModel.addChild(liveData)      Model model      Database myDB      Service myService      model.setColor(GREEN)      model.addChild(myDB)      model.addChild(myService)      myActivity.associate(liveData)      liveData.associate(myService)      myCloud.aggregate(myService)      deploymentView.addChild(view)      deploymentView.addChild(viewModel)      deploymentView.addChild(model) |

Code . Programming Language 형태의 ADL Code

* + 1. 후보구조 . 확장 가능한 Component Type

사용자가 자주 사용되는 Component 형태를 선언하고 이를 통해 중복된 Code를 줄여 보다 간결하게 작성이 가능하다.

후보구조 1에서 정의한 ADL을 기준으로, 아래와 같이 myFragment를 선언하고, 이후에 해당 선언을 Component처럼 사용한다.

|  |
| --- |
| Activity(name=myActivity)  Fragment(name=fragment1, description=”My Fragment”, fillcolor=GREEN, shape=box**) as** **myFragment**  **myFragment**(name=fragment2)  **myFragment**(name=fragment3) |

Code . Component Type 확장

상기와 같이 작성시 아래와 같이 표현될 수 있다.

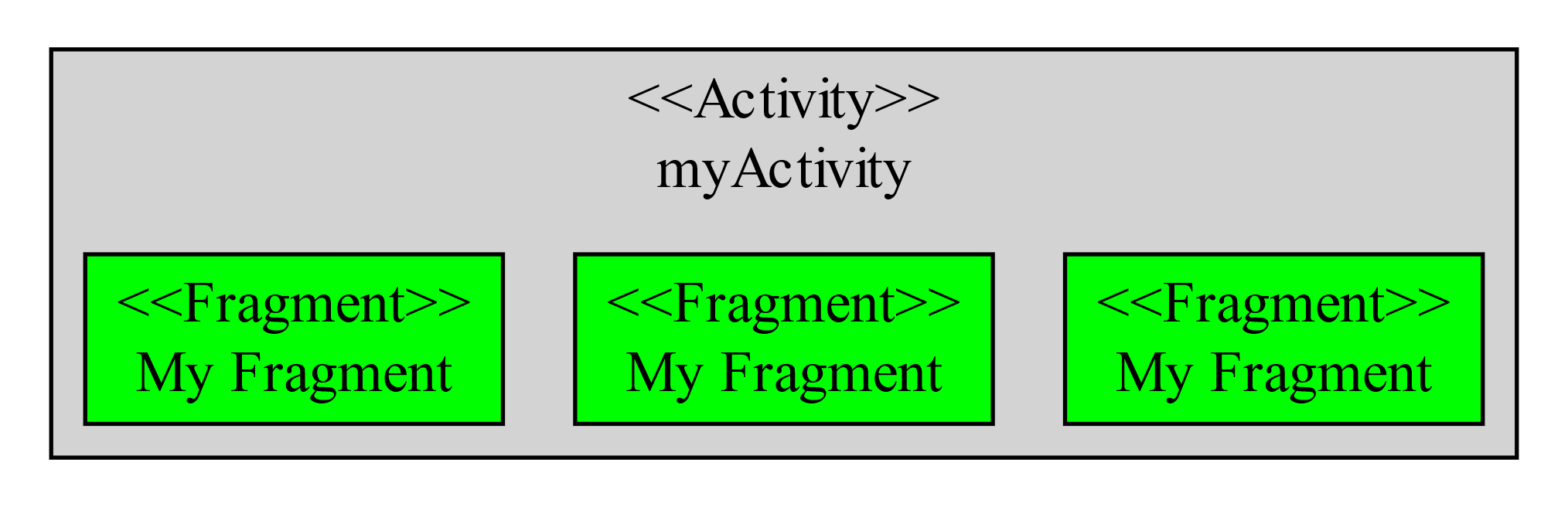


그림 . Component Type 확장

‘as’ 예약 어를 통해서 새로운 Component Type을 선언하고, Translate 과정 중 해당 Component Type을 발견하면 선언 시 사용한Component Type 및 Attribute 속성으로 변환한다. 이를 위해서는 Syntax Tree를 생성하는 시점에 새로 선언된 Component Type에 대한 Tree를 별도로 보관하고, 이후 해당 Type의 Element나 Relation을 인식하면 보관된 Tree를 copy해서 Syntax Queue로 전달한다.

* 1. QA\_02 ADL Code의 Graph Description Language Code 변환 시간

사용자는 ADL Code 작성후에 Graph로 변환시간이 빠를수록 만족도가 높을 것이다. 즉각적인 Graph 반환을 위해서는 Graph Visualization library로 변환 요청하기 전 단계인 Graph Description Language로의 언어 변경이 빨라야 하며 ADL Code를 Graph Description Language Code로 빠른 시간안에 변환하는 후보 구조를 각각 검토한다.

* + 1. 후보구조 . ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계

ADL Code를 Graph Description Language로 변환하는 동작은 어휘분석(lexical analyzer) 단계와 구문 분석(syntax analyzing) 단계, 그리고 의미 분석(semantic analyzer) 단계로 나뉘어진다. Batch Sequential Style로 앞선 작업을 완료하며 순차적으로 실행이 되는데, 이를 Pipe & Filter 형태로 변경해서 각각의 작업을 Thread로 병렬 처리하도록 하면 변환 시간이 단축될 수 있다. 이를 위해서는 Data를 Stream 형태로 발행할 수 있도록, 각 단계에서 분석이 가능한 최소한의 형태로 Data를 구성하는 것이 필요하다.

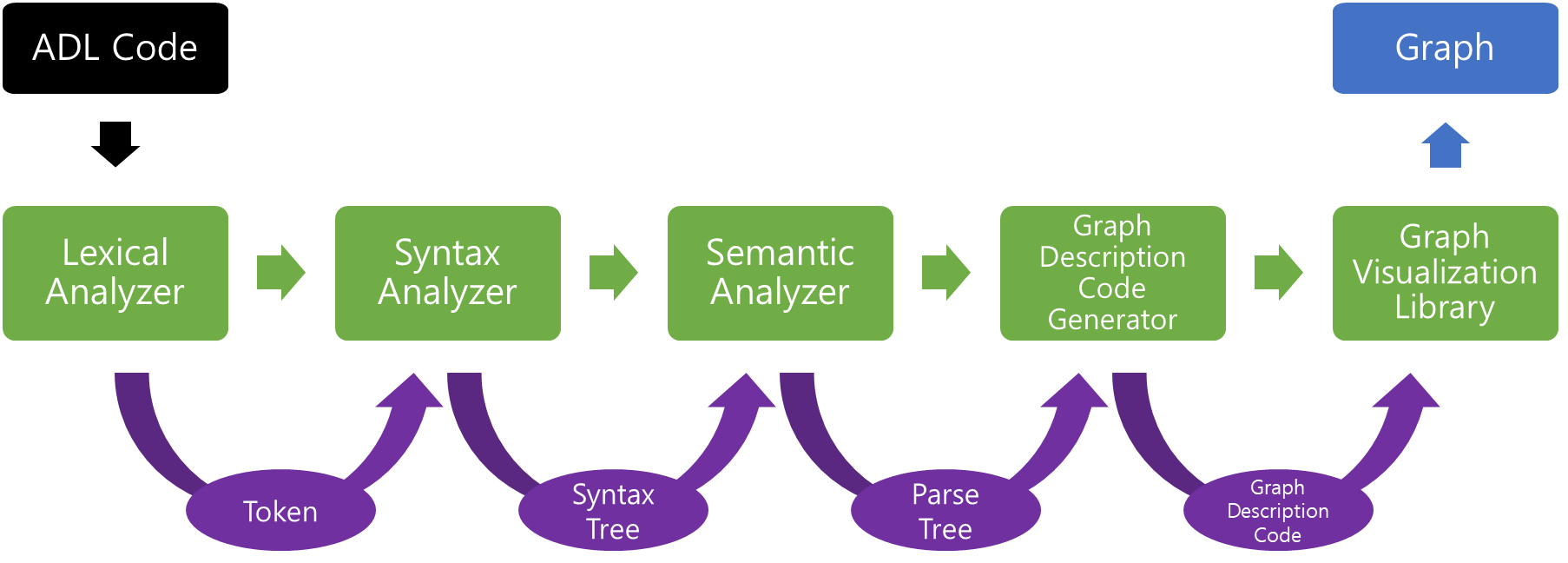
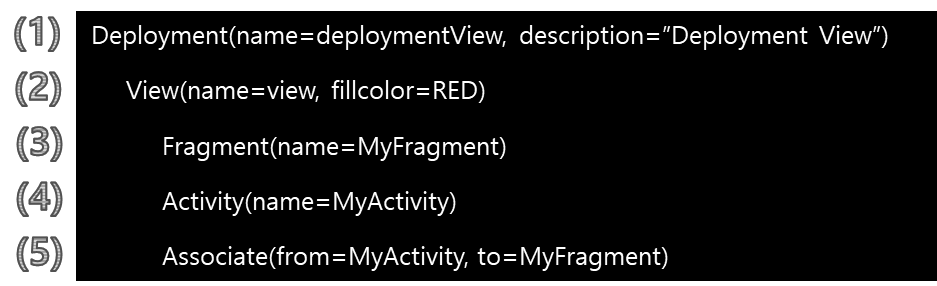


그림 . ADL Code의 Graph 변환 단계

예를 들어 하기 후보구조1에서 정의한 ADL을 변환한다고 할 때 각 line 별로 분석이 가능한 Data가 될 수 있다. 이에 따라 각 Line 별로 상기 분석 단계 flow를 진행하도록 한다.



Code . ADL Code

어휘분석 단계에서는 Tokenizer를 통해 각 문자열을 Token으로 분류하고, 한 Line의 Tokenizing이 끝나게 되면 해당 Token을 바로 구문 분석 Queue로 push하고 어휘분석 Thread는 다음 Line을 Tokenizing 한다.

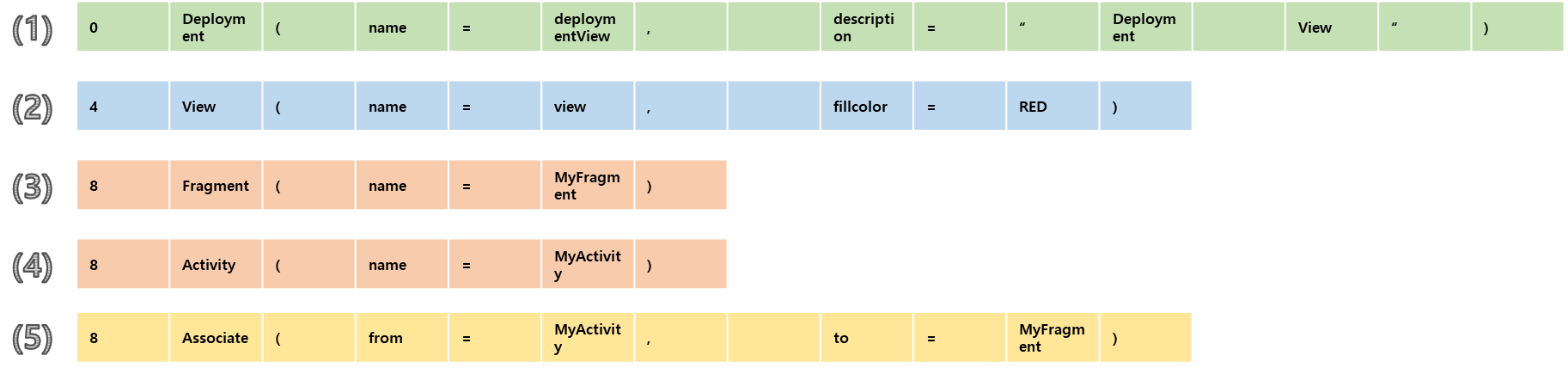


그림 . Token

이후 구문 분석 단계에서는 구문분석 Queue를 통해 전달된 Token에 대한 Syntax Tree를 각 문자열별로 생성하며, Syntax Tree를 생성하면 바로 의미 분석 Queue로 push하고, 다시 구문분석 Queue의 Token을 분석한다.

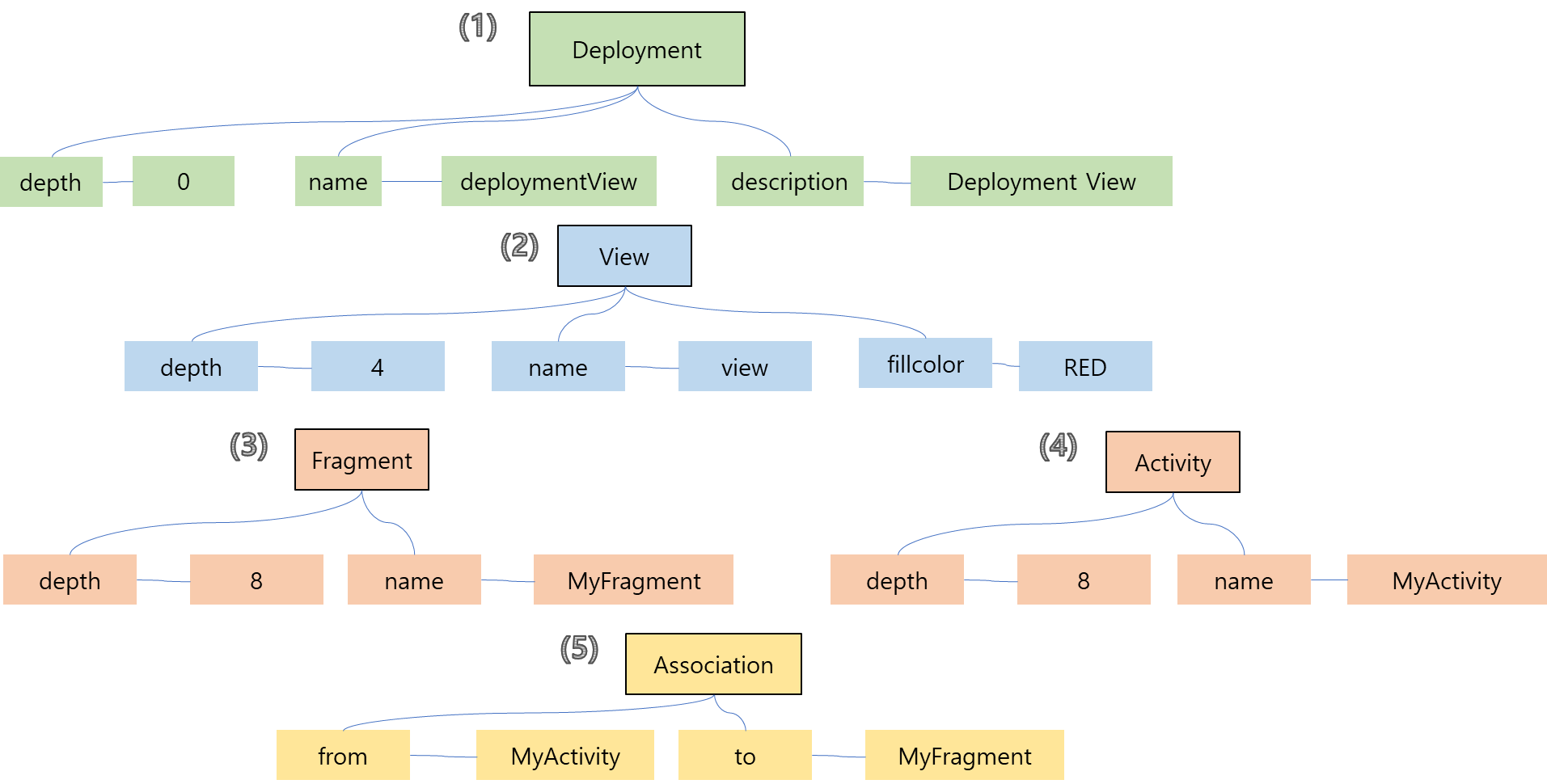


그림 . Syntax Tree

이후 의미 분석 단계에서는 의미 분석 Queue를 통해 전달된 Syntax Tree를 depth를 고려해서 Parse Tree에 추가하며 각 Tree의 연관관계를 추가 변경 구성한다. 그리고 의미 분석 Queue를 통해 전달되는 Syntax Tree를 지속적으로 분석한다.

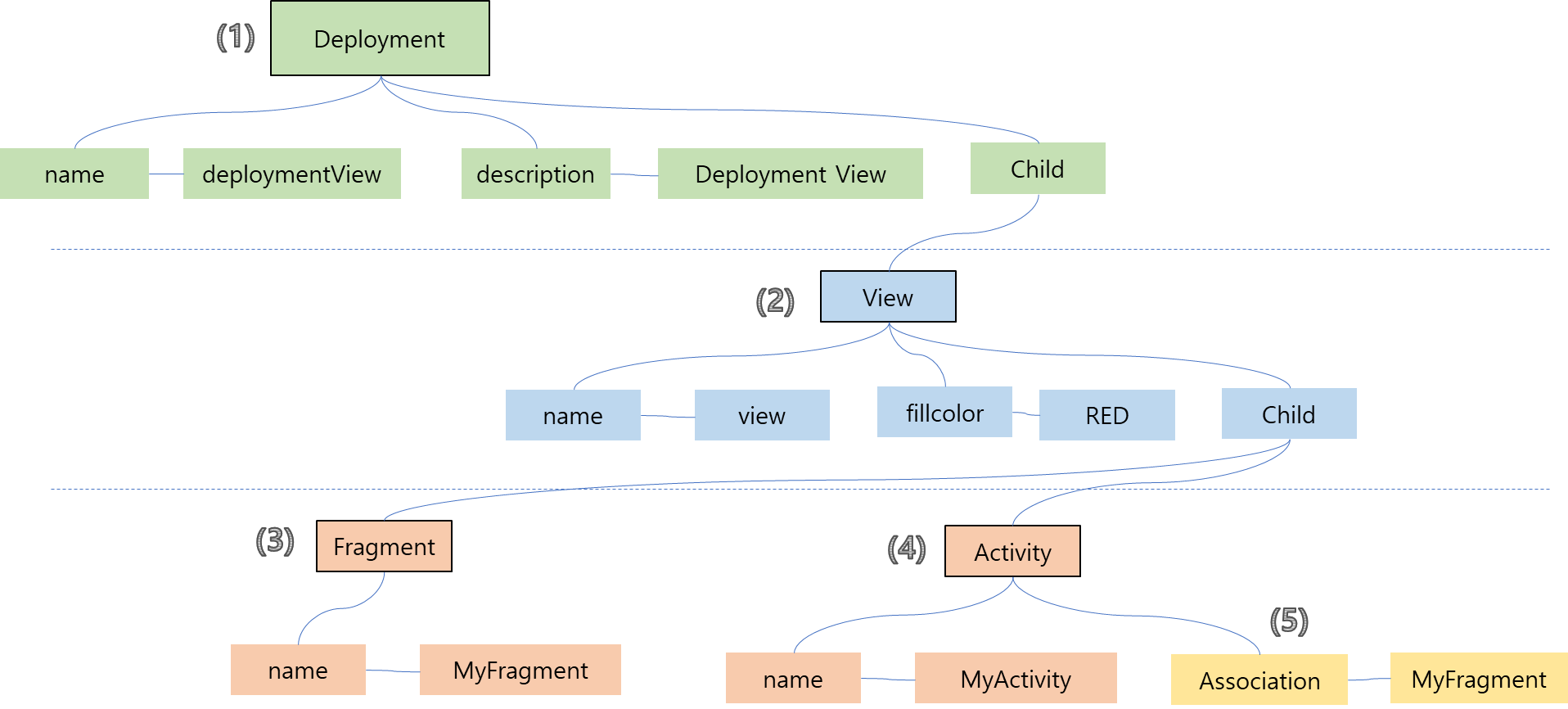


그림 . Parse Tree

Parse Tree가 완성되면 Graph Description Code Generator를 통해 Graph Visualization Library가 인지할 수 있는 Graph Description Code를 생성한다. Parse Tree는 각 Element의 Attribute 및 계층 관계가 모두 표현되어 있어서, Parse Tree를 통한 Graph Description Language 변환은 해당 Language에 대응하는 단순 Mapping으로 변환이 가능하다.

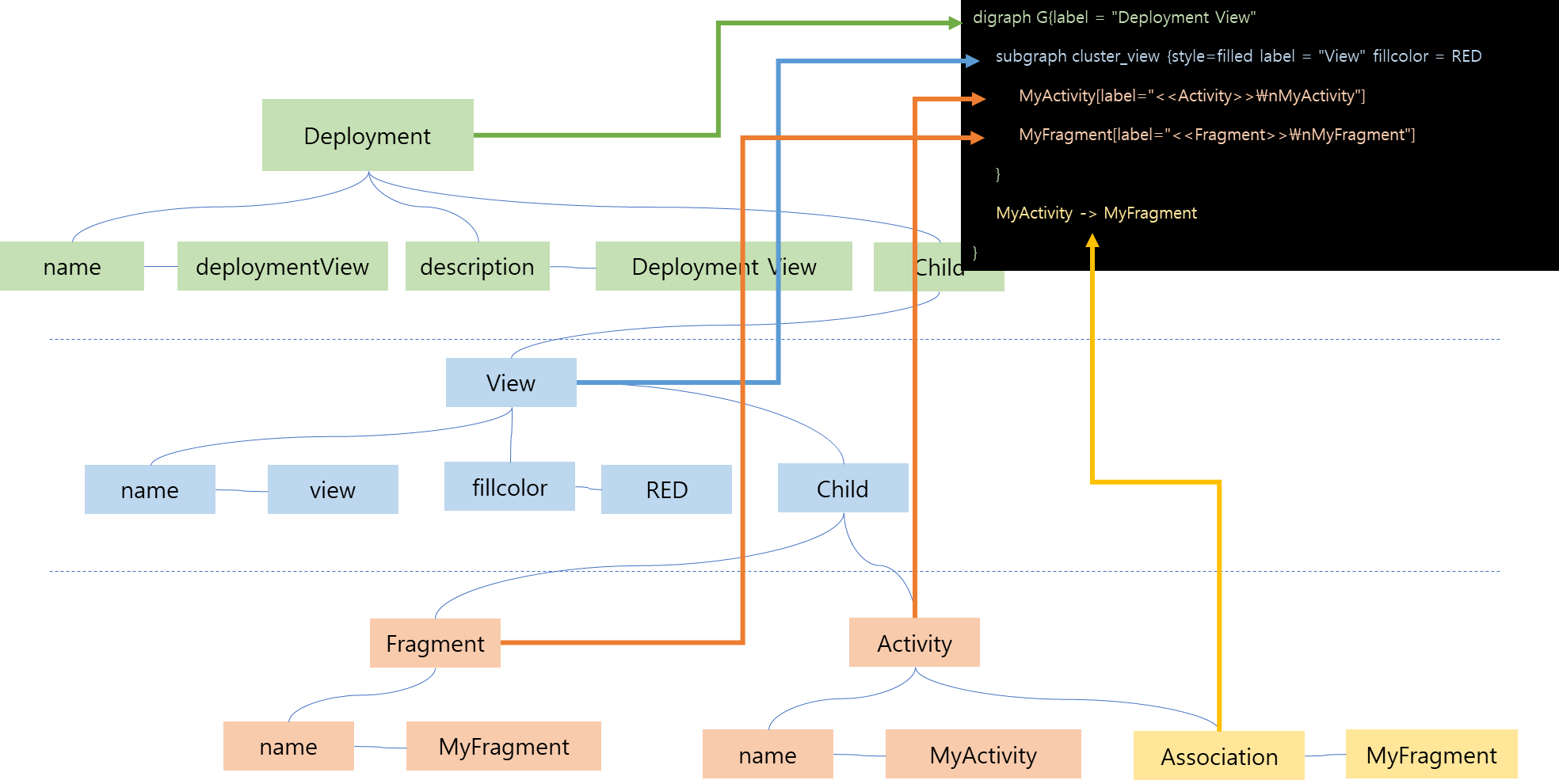
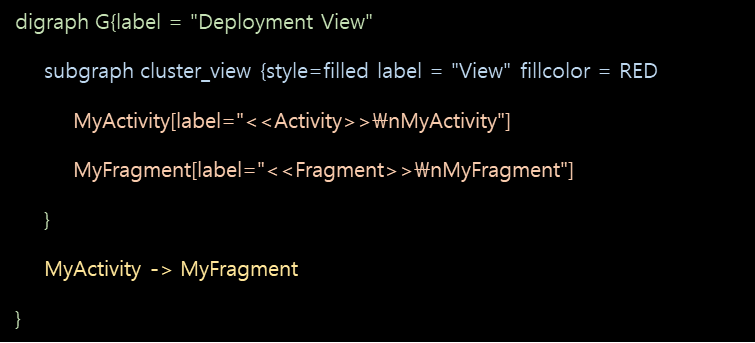


그림 . DOT Graph Description Language Code

예시 ADL Code를 DOT Graph Description Language로 변환한 Code는 아래와 같다.



Code . DOT Graph Description Language Code

생성한 DOT Code를 Graphviz Graph Visualization Library를 통해 Graph 변환을 요청하면 하기와 같은 Graph를 얻게 된다.

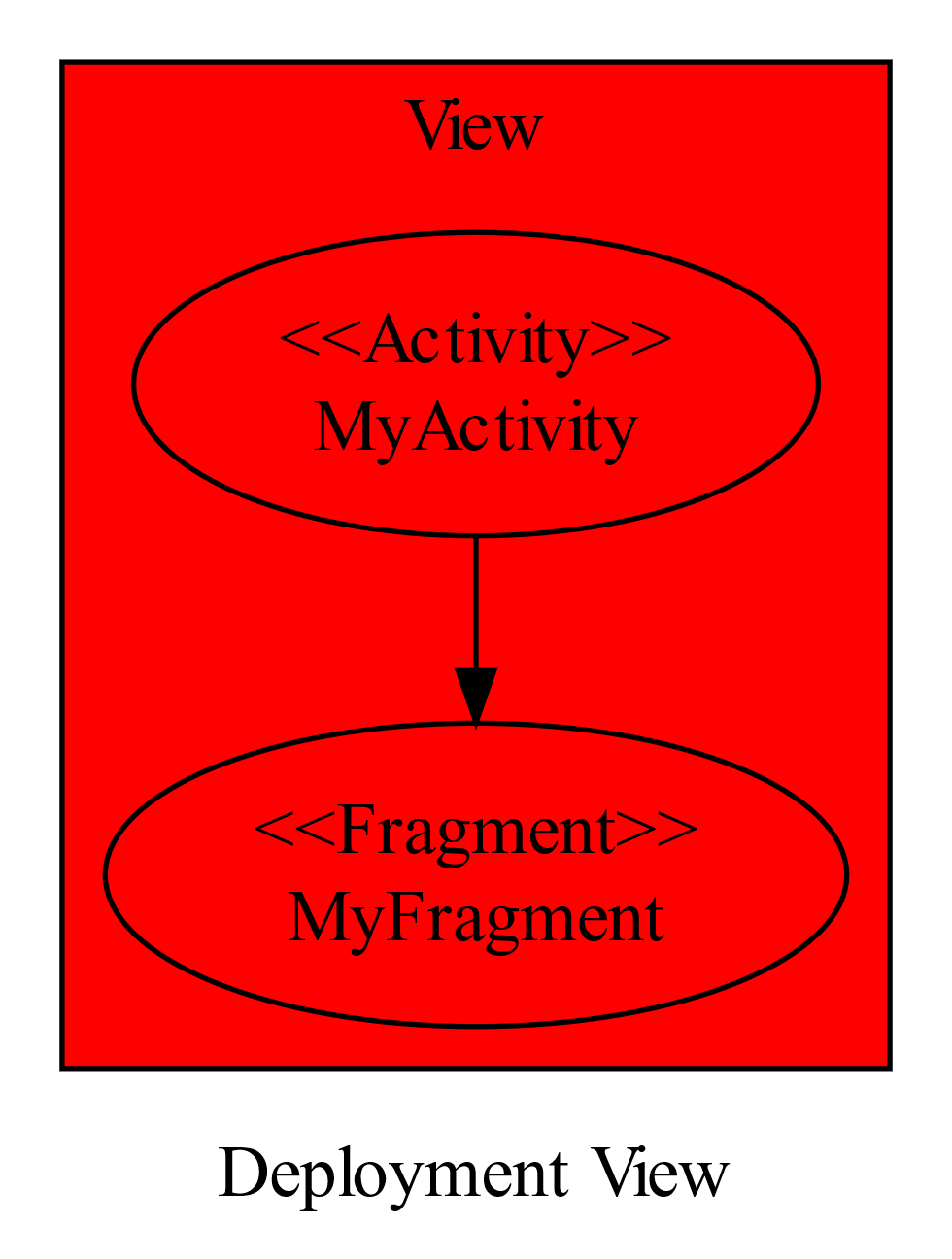


그림 . Graphviz Graph

* + 1. 후보구조 . Update를 위해 Token 및 Parse Tree 유지

사용자가 Translate를 진행하고 나서 Code를 수정하고 다시 Translate를 요청하는 것은 빈번한 사용 방법일 것이다. 이에 변환 효율을 높이기 위해 Token및 Parse Tree를 소멸시키지 않고 Memory상에 유지한다.

1. 다시 Translate시 기존 Token과 비교를 해서 추가 / 삭제 / 변경된 Token만 구문 분석 Queue로 전달하고
2. 구문 분석을 통해 Syntax tree를 생성 후 의미 분석 Queue로 전달하고
3. 의미 분석 단계에서 Syntax tree의 depth와 line을 확인해서 Parse tree의 적합한 위치에 추가 / 삭제 / 변경을 진행

한다.

Token에 Code의 Line number를 추가하고 이를 바탕으로 0 depth Element부터 재귀적으로 depth와 line number를 비교하며 대상 element를 찾아낸다. 추가 / 삭제의 경우 depth와 line number가 같은 경우 해당 Element를 변경 / 삭제하고, 추가의 경우 Token의 보다 한 depth 얕은 Element 중에 Line Number가 Token의 Line number보다 크지 않으면서 가장 가까운 Element에 Child Element로 추가 한다.

* + 1. 후보구조 . Token, Tree 구성을 위한 Resource 선 할당

성능을 위해 각 Token 및 Tree객체를 미리 Memory 상에 allocate해둔 후에 변환 작업을 수행한다면 Memory allocate를 위한 시간 소모를 없앨 수 있을 것이다. 하지만 사용하지 않는 메모리 점유를 높여서 전체 Resource 효율에 악영향을 끼칠 수 있다. 이에 따라 일반적인 View 구성을 위한 Component 수를 산정하고 이에 해당하는 만큼만 Memory 점유를 한다면 큰 손실없이 성능을 개선할 수 있다. Android Architecture는 기본적으로 Layered형태로 작성이 많이 되며 각 Layer별 Component 수는 일반적으로 10개 내외로 5개의 Layer를 둔다고 하면 50개 정도의 Syntax Tree가 생성될 수 있다. 또한 후보구조 1에 정의된 ADL로 Code 작성시 한 line의 최대 token 개수가 30개를 넘지 않는다. 이에 맞게 Translate시 이미 allocate된 Memory pool에서 Token 및 Tree를 구성하고, 해당 Pool의 용량을 넘어서게 되면 추가 할당하는 구조를 통해 성능을 향상시킨다.

* + 1. 후보구조 . Tree 구성을 위한 Resource 효율화

Translate 중에 사용하는 Memory를 최적화해서 성능을 향상시킬 수 있다. Token을 통해 Syntax Tree를 만들고, Syntax Tree를 통해 Parse Tree를 만드는 과정은 결국 Token Data의 의미를 분석해서 배치를 하는 방법이다. 이때 공통적으로 사용되는 Token Data를 공유 데이터 영역 Repository로 구성하고 각 Tree Node의 Value값을 별도로 할당하지 않고 Repository의 Token Data를 Reference 하는 방식으로 Tree를 구성한다. Tree Node의 Value를 위해 별도의 Memory를 할당하지 않아도 되기 때문에 효율적으로 Resource를 사용하게 되고 성능 또한 향상될 수 있다.

* + 1. 후보구조 . Dot language Parser library를 사용

Python과 같은 기존 Programming language로 Code를 작성하면 Dot language로 변환할 수 있는 library들이 존재한다.([gvgen](https://github.com/stricaud/gvgen)) 해당 Library를 사용할 경우 Programming language를 그대로 사용할 수 있어서 Library의 사용 방법을 익히는 것 외에는 익숙한 Programming language를 그대로 사용할 수 있는 이점이 있다.

* 1. QA\_03 Translate 요청 시 첫 Graph 반환 시간

사용자가 Translate 요청 시 Code양이 많은 경우 Grape변환에 시간이 많이 소요되게 된다. 이 경우 즉각적인 반응이 이루어지지 않은 상황이어서 사용성에 문제가 생기며 Translate 요청 시 얼마나 빨리 첫 Graph를 반환하는지가 중요한 품질 속성이 되겠다. 이에 따라 Graph를 가능한 빨리 반환할 수 있는 후보구조를 알아본다.

* + 1. 후보구조 . 연속적으로 Graph를 갱신

Translate 동작은 크게 1. ADL을 분석하는 과정, 2. 분석된 ADL을 DOT과 같은 Graph Description Language로 변환하는 과정, 3. Graph Description Code를 Graph로 변환하는 과정이 있다. 이중 3에 해당하는 과정은 본 시스템 영역 밖의 동작으로 Graph Visualization Library 자체의 동작이어서 고정된 출력시간이라 볼 수 있다. 이에 따라 개선이 가능한 구간은 1, 2번 구간인데, 1, 2번 구간이 모두 완료되고 3번 과정을 수행하게 되면 ADL Code양에 따라 Graph 반환 속도가 차이가 발생한다.

현재는 1번 과정을 모두 완료후에 2번 과정이 수행되고 있는데, 1번 과정을 완료하기 전에 변환된 Parse Tree가 생기는 대로 2번 과정을 수행하고, 이를 통해 생성된 Graph Description Language Code를 Graph Visualization Library에 변환 요청하여 Graph를 계속해서 update해준다. 이를 위해서는 2번 과정(분석된 ADL을 Graph Description Language로 변환하는 과정)도 1번 과정의 각 분석단계처럼(어휘 분석, 구문 분석, 의미 분석) Stream 형태로 변환이 지속적으로 이루어지도록 해야 한다.



그림 . 연속적인 Graph 갱신

* + 1. 후보구조 . 일정한 주기로 Graph를 갱신

후보구조 10은 Parse Tree가 변경될 때 마다 새로운 Graph를 반환하도록 한다. 이경우 빈번한 Graph 변환 요청으로 인해 시스템의 전체 성능에 영향을 끼칠 수 있다. 사용자의 사용성에 있어서는 즉각적으로 반응이 일어나는 것이 중요한 부분이지, 매 라인별로 update되는 graph를 확인하는 것은 그다지 중요하지 않다. 이에 따라 일정한 간격을 두고 Graph의 update를 진행한다.



그림 . Schedule Queue에 의한 Graph 갱신

처음 Schedule Queue에 Parse Tree가 전달되면 해당 시간을 기억해두고 바로 Graph Description Language Code 생성 및 Graph 변환을 진행한다. 이후 update되는 Parse Tree는 Graph Description Code Generator로 바로 전달하지 않고 일정시간(500ms ~ 1000ms) 후에 마지막 Parse Tree만 전달함으로써 Graph 변환 과부하를 막을 수 있다.

* + 1. 후보구조 . Translate 요청전에 Background parsing

Translate 요청이 오기전에 Background에서 Graph Description Language Code 변환 작업을 하게 되면 Translate 요청 시 즉각적인 Graph 반환이 가능하다. 이를 위해서는 ADL 도구에서 Code변경이 발생할 때마다 ADL Framework으로 Background Translate 요청을 해야 하며, ADL Framework에서는 해당 요청에 대한 Graph를 반환하지 않고 Memory상에 유지하고 있다가 Translate 요청이 발생할 때 별도로 ADL 변환 작업을 하지 않고 Memory상에 기억해 두었던 Graph를 전달한다.

* 1. QA\_04 Element Type 변경 용이성

Element Type은 본 시스템이 정의한 Android Application에 맞게 ‘Activity’, ‘Service’, ‘Content Provider’, ‘Broadcast Receiver’, ‘Fragment’ 및 Architecture 요소인 ‘Model’, ‘View’, ‘ViewModel’, ‘Controller’, ‘Presenter’, 일반적인 Component인 ‘Database’, ‘Server’, ‘Client’ 등으로 정의되어 있다. 하지만 이 Type은 사용자 요청에 의해 지속적으로 추가 / 삭제 / 변경될 수 있다. 더불어 Android App 뿐 만 아니라 iOS, Windows등 다른 Platform App을 위한 ADL Framework으로도 확장될 수 있다.

Element Type을 사용하는 Module은 Syntax Analyzer(Syntax Tree를 구성할 때), Code Generator(Class 생성시), Autocomplete(Guide Component 생성시) 로써 추가 / 삭제 / 변경 시 해당 모듈의 큰 변경 없이 적용이 가능해야 하며 이에 용이한 후보구조를 검토한다.

* + 1. 후보구조 . 별도의 AppModel library로 지연 바인딩

Application Model에 의존적인 부분을 구분하여 Element Type을 AppModel로 정의하고 Concrete Module을 ADL Framework에서 분리한다. AppModel의 추가 / 삭제 / 변경이 필요한 경우에 ADL Framework을 다시 배포할 필요없이 AppModel library만 배포함으로써 적용이 가능하다. AppModel을 참조하기 위한 Interface는 ADL Framework이 그대로 가지고 있기 때문에 ADL Framework의 변경사항 없이, 변경이 필요한 AppModel library의 수정만으로 적용이 가능하다. ADL Framework 구동 시 Concrete module Library에 동적으로 바인딩한다.



그림 . AppModel의 Library 분리

IAppModel은 아래와 같은 interface로 구성되어 있으며, 해당 API를 통해 ADL Framework에서 지원하는 Element를 가져올 수 있다.

|  |  |  |
| --- | --- | --- |
| Method | Return | Description |
| getViewType | ArrayList<String> | View의 종류를 String 형태의 Array로 return |
| getSubsystemType | ArrayList<String> | Subsystem의 종류를 String 형태의 Array로 return |
| getComponentType | ArrayList<String> | Component의 종류를 String 형태의 Array로 return |
| getAllType | ArrayList<String> | 해당 AppModel에서 정의하는 모든 Element들을 return 한다. |
| getClassName(element) | String | 특정 Element의 실제 Platform상의 full class name을 return한다.  Ex) IAppModel.getClassName(“Activity”)  Return : “android.app.Activity” |

표 . IAppModel Interface

* + 1. 후보구조 . Element Type에 Decorate Pattern을 적용

Element Type에 Decorate Pattern이 적용되어 있으면, Type의 확장이 용이하다. Concrete Module인 Element Type의 변경 없이 Element Type을 추가할 수 있으며 이를 통해 다양한 Platform에 추가되는 Type을 용도에 맞게 변경할 수 있다.



그림 . Element Type with Decorated pattern

* + 1. 후보구조 . Element Type에 Strategy Pattern을 적용

Element Type에 Strategy Pattern이 적용되어 있으면, 상황에 맞게 Element Type을 변경하며 사용이 가능하다. ADL Framework에서 Element Type을 설정할 수 있는 API를 제공해주고 이에 따라 설정된 Element Type을 변경하며 사용할 수도 있다. Element Type추가도 기존 Module의 수정 없이 적용이 가능하며, 변경 / 삭제도 해당 Module 내에서만 수정이 이루어진다.



그림 . Component Type with Strategy pattern

* 1. QA\_05 Graph Visualization Library의 변경 용이성

Translate 과정에서 Graph Visualization Library에 의존적인 Flow가 존재한다. 의미 분석까지 완료한 Parse Tree를 기준으로 1. DOT과 같은 Graph Description Language(GDL) Code로 변경하는 부분과 2. Graphviz와 같은 Graph Visualization Library를 통해 해당 GDL을 Graph로 변환하는 부분이다. Graph Visualization Library들은 자체적으로 분석이 가능한 Grape Description Language(GDL)을 가지고 있으며, (ex)Graphviz는 DOT 등, Neo4j는 Cyper, AbsInt는 aiSee) 이에 따라 Graph Visualization Library에서 해석이 가능한 GDL로 변경이 필요하다.

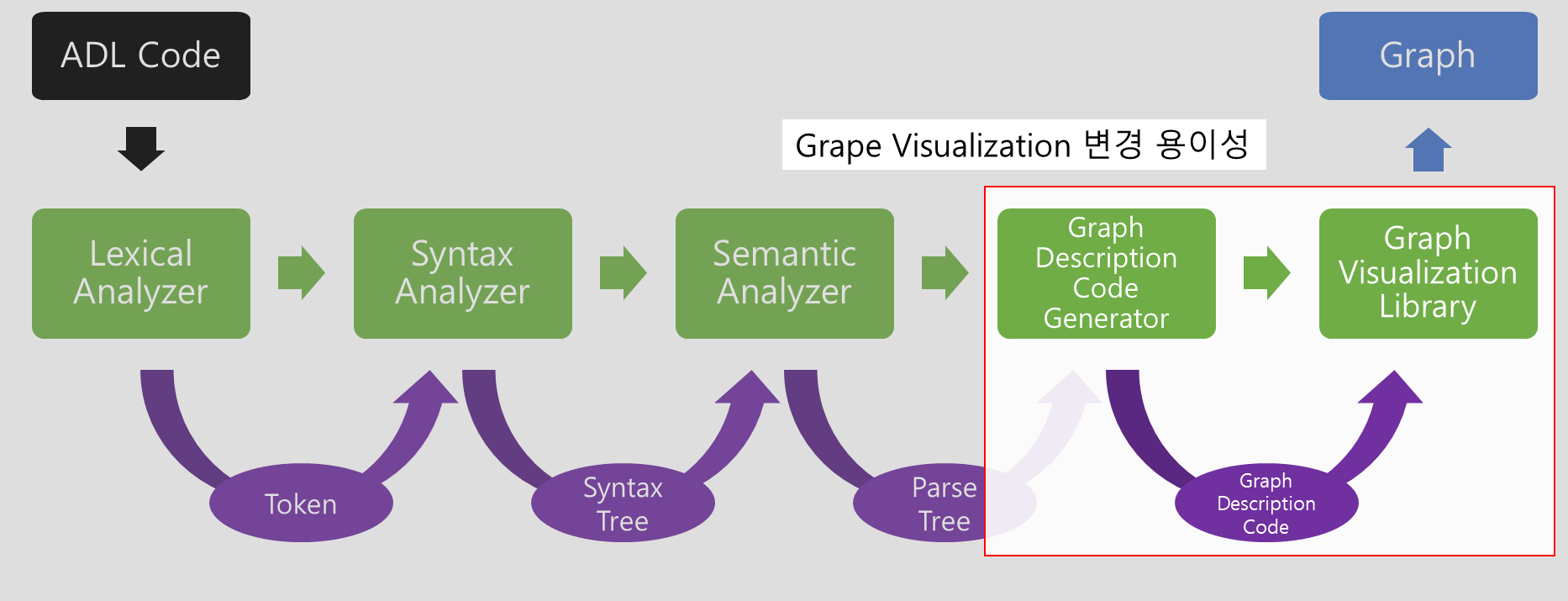


그림 . Graph Visualization 변경 용이성

위와 같은 상황에서 Graph Visualization Library를 변경할 때 다른 모듈에 영향을 최소화할 수 있는, 변경이 용이한 후보구조를 검토한다.

* + 1. 후보구조 . 별도의 GraphDrawer Library로 지연 바인딩

ADL Framework에서 Graph Visualization Library에 의존 적인 부분을 IGraphDrawer로 선언하고 이를 위한 Concrete Module은 별도의 GraphDrawer Library로 분리한다. GraphDrawer Library는 별도로 배포함으로써 ADL Framework 수정없이 변경이 가능하며, ADL Framework 구동 시에 Concrete module library에 동적으로 바인딩한다.

Semantic Analyzer는 생성된 Parse Tree를 IGraphDrawer를 통해 전달하고, GraphDrawer에서는 전달받은 Parse Tree를 자체 GDL로 변환하는 작업과 변환된 GDL을 Graph로 변환하는 두가지 작업을 수행한다. Graph로 변환이 완료된 이후에는 등록되어 있는 Graph Listener로 결과를 반환한다.



그림 . IGraphDrawer Library

IGraphDrawer에서 제공하는 API는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| Method | Return | Description |
| translate(token, ParseTree) | void | ParseTree를 전달하며 Graph 전환 요청 |
| registerListener(GraphListener) | Boolean | Graph 전환 완료 시 Graph를 전달받을 Listener를 등록한다. |
| unregisterListener(GraphListener) | void | Graph 전환 완료 Listener를 해지한다. |

표 . IGraphDrawer API

Graph 변환 결과를 받는 IGraphListener는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| Method | Return | Description |
| onComplete(token, Graph) | void | Graph 변환 완료 시 Graph를 전달 |
| onError(token, Cause) | void | Graph 변환 중 Error 발생시 Error Cause를 전달 |

표 . IGraphListener API

* + 1. 후보구조 . GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용

Parse Tree를 Graph Description Language(이하 GDL) Code로 변경하는 모듈과 해당 GDL Code를 Graph로 생성하는 부분을 분리하고 각각의 구현부와 추상층을 분리하여 변경이 용이하도록 구성한다.



그림 . GDL Code 생성과 Graph 생성 분리된 Bridge Pattern

Semantic Analyzer가 Drawer로 ParseTree와 함께 translate 요청하면, 동적 바인딩된 Drawer가 자신의 GDL Code로 변경을 하고 IGraphDrawer을 통해 변경된 GDL Code를 Graph로 translate 요청한다. 이에 따라 하나의 GDL Code를 다양한 Graph Visualization Library로 표현이 가능하다.

* 1. QA\_06 Autocomplete 완료시간

ADL 문법이 간단하게 정의되어 있다고 하더라도, 실제 코딩할 때 암기를 해야 할 것들이 존재한다. 어떤 예약된 Element들이 존재하는지, Attribute는 어떤 항목들이 있는지 등 익숙해지더라도 실수하기 쉬운 부분들이 있으며, 본 시스템은 이런 실수를 방지하기 위해 자동완성 기능을 제공한다. 하지만 기능을 온전히 활용하기 위해서는 Autocomplete 요청에 대한 빠른 응답이 필수적이며 이를 위한 후보구조를 검토한다.

* + 1. 후보구조 . Radix Tree로 Keyword 관리

ADL에서 정의된 예약어 들은 크게 4가지 종류이며, Element Type, Element Attribute, Relation Type, Relation Attribute이다. 각각의 종류별 요소들을 ADL Framework 구동시점에 모두 읽어오고 이에 해당하는 Radix tree를 생성한다. ADL 도구에서 Autocomplete 요청이 들어올 때 이를 활용하여 문자열 배열을 반환하며, Radix Tree의 특성대로 문자열 비교를 최소화하여 빠른 응답 성능을 확보할 수 있다.

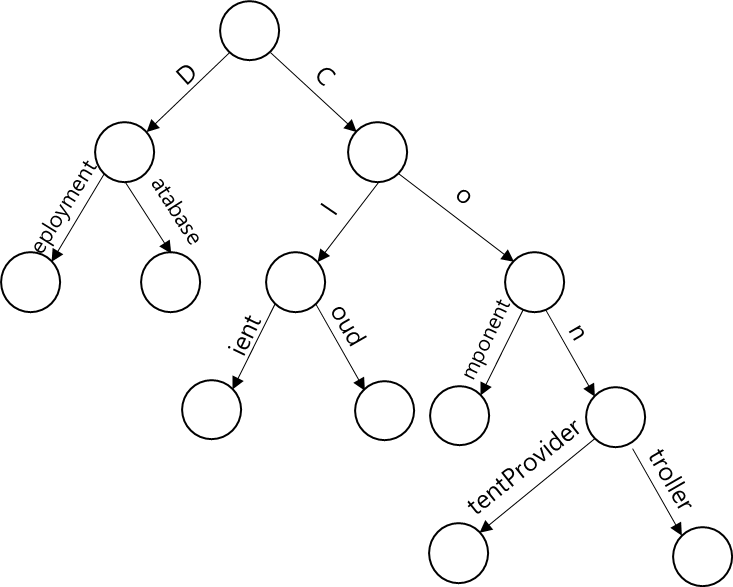


그림 . Keyword Radix Tree

* + 1. 후보구조 . 문맥에 따른 Autocomplete 적용

후보구조 1에 따르면 먼저 Element Type이나 Relation Type을 기입하고 이후에 각 Type에 맞는 Attribute를 기입한다. 즉 문맥에 따라 Element Type이나 Relation Type를 추천해야 할 때와 각 Type에 맞는 Attribute를 추천해야 할 때가 구분된다. ADL 도구에서 현재 입력하고 있는 문자와, 해당 Line에 있는 전체 문자들을 전달하면, 이를 분석해서 어떤 Type을 추천해야 하는 상황인지 판단할 수 있다. Pipe & Filter 구조는 Filter를 교체하며 원하는 결과를 얻을 수 있을 수 있는 변경용이성이 우수한 구조이다. ‘후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계’에서 Syntax Analyzer를 Autocomplete 용도에 맞게 Autocomplete Syntax Analyzer로 변경하고 해당 Analyzer에서는 Token상에 ‘(‘가 존재하는지, 존재한다면 ‘(‘token 이전의 token은 Element Type인지 Relation Type인지, 문맥을 판단하고 이를 Autocomplete Task로 전달한다. Autocomplete Task는 문맥에 따라 제한된 Boundary에서 검색을 하게 돼서 검색의 성능면에서 이점이 있다.



그림 . Autocomplete Syntax Analyzer로 Syntax Analyzer Filter 변경한 Sequence Diagram

* + 1. 후보구조 . ADL 도구에 Keyword Array 전달

시스템에서 인지하는 예약어를 ADL Framework API를 통해 ADL 도구에 제공해준다. ADL 도구는 사용자가 문자 입력 시 해당 예약어를 통해 Autocomplete 힌트를 제공하거나, Dropbox 형태로 가능한 Type을 선택하는 등의 동작을 할 수 있다. ADL Framework 자체적으로 Autocomplete 기능을 관리하는 것보다 각 EditText를 직접 Control하는 ADL 도구에서 예약어 List를 가지고 필요에 따라 활용하면 빈번한 ADL Framework으로의 Method Call을 줄일 수 있어서 보다 효율적일 수 있다.

* 1. QA\_07 ADL Framework API의 확장성

본 시스템의 요구사항에 따라 ADL Framework이 ADL 도구를 위해 제공하는 API는 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| API | Return | Sync | Description |
| translate(ADL Code) | Graph image | Async | ADL Code를 Graph로 변환 |
| autocomplete(prefix String) | 문자열 배열 | Sync | Prefix String으로 시작하는 예약어 배열을 반환 |
| codegenerate(ADL Code) | Void | Async | ADL Code에 따른 Skeleton Class File을 생성한다. |

표 . ADL Framework API

하지만 QA중 아래와 같은 변경용이성이 추가로 요구된다.

* QA\_04. Element Type 변경 용이성
* QA\_05. Graph Visualization Library의 변경 용이성
* QA\_08. Code Generation의 변경 용이성

해당 QA를 달성하기 위한 Architecture들이 존재하나 Architecture를 변경이 용이하게 구성하였다고 하더라도 사용자가 각각을 변경할 수 있는 Interface의 변경 또한 필요하다. 이와 같은 ADL Framework에서 ADL 도구를 위해 제공하는 API의 추가가 기존 모듈에 큰 영향없이 추가될 수 있어야 하며 이를 위한 후보구조를 검토한다.

* + 1. 후보구조 . Interface Layer를 분리 Publish – Subscriber Pattern 적용

ADL Framework에서 Interface Layer를 분리하여 독립적으로 개발 발전시킨다. Interface Layer와 Core Layer를 분리하기 위해 Publish – Subscriber Pattern을 적용한다. 변경이 많이 발생할 수 있는 Interface Layer와 비교적 변경이 적게 발생할 Core layer를 분리하고, interface layer가 Core Layer로 dependency를 가지도록 설계한다. Core layer로부터의 응답은 Task Observer의 Concrete Module을 등록해서 받을 수 있고, ADL 도구도 마찬가지로 ADL Listener의 Concrete Module를 등록해두는 방식으로 응답을 받을 수 있다.



그림 . Layered, Publish – Subscriber Architecture

* + 1. 후보구조 . Task 기반의 Event Bus 적용

Event Bus를 두고 Task 기반으로 Interface Layer와 Core Layer가 Communication할 수 있도록 설계한다. 새로운 기능이 추가되면 Interface Layer에서는 해당 API를 ADL Manager에 추가하고 단순히 Event Bus에 Event를 등록하게 된다. Core Layer에서는 해당 Event를 수신하는 새로운 Task를 생성하고 해당 Task 내에서 신규 기능을 구현하게 된다. 또한 Return값도 Event Bus를 통해 전달되기 때문에 신규 Task에서는 결과값을 Event bus로 전달하기만 하면 되며, Interface Layer에서는 해당 Event를 수신해서 결과값에 해당하는 동작을 수행하면 된다. Interface layer와 Core Layer가 독립적으로 개발될 수 있어서 변경이 용이하다 할 수 있다.



그림 . Event Bus Architecture

* 1. QA\_08 Code 생성 Language 확장성

본 시스템은 Deployment view와 Component & Connector view를 기준으로 기본적으로 Java Skeleton code 생성이 가능하다. 하지만 Android Application은 현재 Java 뿐 아니라 Kotlin, Flutter 등으로도 개발이 되고 있다. 추후 언제든지 Programming Language가 추가될 수 있는 상황이며 이에 따라 Programming Language 추가가 용이한 구조후보구조를 검토한다.

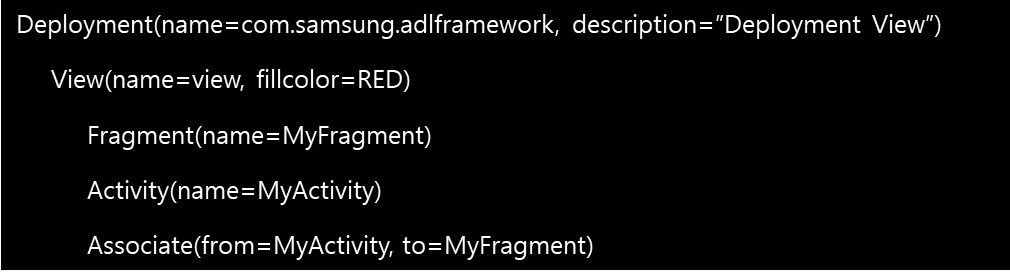
* + 1. 후보구조 . Code Generation Library 적용

Code Generation동작을 별도 Library에 위임한다. ADL Framework에서는 변환 요청과 함께 Parse Tree를 Code Generation Library에 요청한다. 해당 Library에서 Language에 따라 Parse Tree를 바탕으로 Code Generation 한다. 이 경우 Element Type별 Mapping Class Table이 필요한데, 이에 따라 후보구조 13과 같은 IAppModel에 의존성이 있다.



그림 . Code Generation Library

예를 들어 아래와 같은 후보구조 1과 같은 방식의 ADL Code가 있다면 Element Type에 해당하는 Class를 인지하기 위해 Element Type Table이 필요하다.



Code . Code Generate를 위한 ADL Code

|  |  |  |
| --- | --- | --- |
| Element Type | Value | Class Name |
| View | ComponentAndConnector  Deployment | -  - |
| SubSystem | Model  View  ViewModel  Controller  Presenter | -  -  -  - |
| Component | Activity  Service  Broadcast Receiver  Content Provider  Fragment  Database  Component  Process  Thread | **android.app.Activity**  **android.app.Service**  **android.content.BroadcastReceiver**  **android.content.ContentProvider**  **android.support.v4.app.Fragment**  **android.database.sqlite.SQLiteDatabase**  -  -  - |
| External Component | Server  Client  Cloud | -  -  - |

표 . Element Type Table

Code Generation Library는 상기 Table을 기준으로 Element Type에 해당하는 class Name이 존재하는 경우 해당 Class를 extends하여 name attribute에 해당하는 이름으로 Class를 생성하며, view나 subsystem의 경우는 아래와 같은 package(directory)를 생성한다.



그림 . Code Generation

* + 1. 후보구조 . Language별 Class Prototype변경

기본적으로 android는 package명과 directory 구조가 같으므로 이는 Language 변경에 따라 차이가 발생하는 부분이 아니며, Language 변경 시 영향을 받는 부분은 각 Language의 Class 선언 시 문법 차이 정도이다. 이를 위해 Strategy Pattern을 적용하고 설정에 따라 Class Prototype을 다르게 설정하여 기존 Module의 변경없이 새로운 Language를 추가한다.



그림 . Language별 Prototype

Java와 Kotlin의 Code 생성시점의 차이점을 비교해보면 상속과 변수 선언하는 방식이 다를 뿐이다. 이에 따라 Class 생성을 위해서는

1. Deployment view나 Component & Connector view의 Name을 ClassPrototype에 전달하여 package 선언 문구를 받아오고,   
   - ex) “package com.samsung.adlframework.view;”
2. Element Type을 ClassProtytype에 전달하여 Type에 해당하는 Class import 선언 문구를 받아오고,   
   - ex) “import android.support.v4.app.Fragment;”
3. 현재 Element의 Name 및 Type을 ClassPrototype에 전달하면 Type에 해당 하는 Class를 AppModel로부터 확인 후 해당 Class를 상속받은 형태의 Name Class 선언 문구를 받아오고,   
   - ex) “public class MyFragment extends Fragment {}”
4. Relation의 to에 해당하는 Name을 ClassPrototype에 전달하여 Member 변수 선언 문구를 받아오고 이를 from에 해당하는 name의 Class에 추가한다.   
   - ex) “MyFragment myFragment”

ClassProtoType의 API는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| API | Return | Description |
| getPackageProtoType(Name) | Package 선언 String | Name에 해당하는 package를 선언하는 문구를 전달한다. |
| getImportPrototype(Type) | Import 선언 String | Type에 해당하는 Class name을 AppModel의 Element Type Table로부터 확인하고 이를 import하는 문구를 전달한다. |
| getClassPrototype(Name, Type) | Class 선언 String | Type에 해당하는 Class name을 AppModel의 Element Type Table로부터 확인하고 이를 상속받는 Name Class를 선언하는 문구를 전달한다. |
| getFieldPrototype(Name) | Member Field 선언 String | Name Type의 변수를 선언하는 문구를 전달한다. |

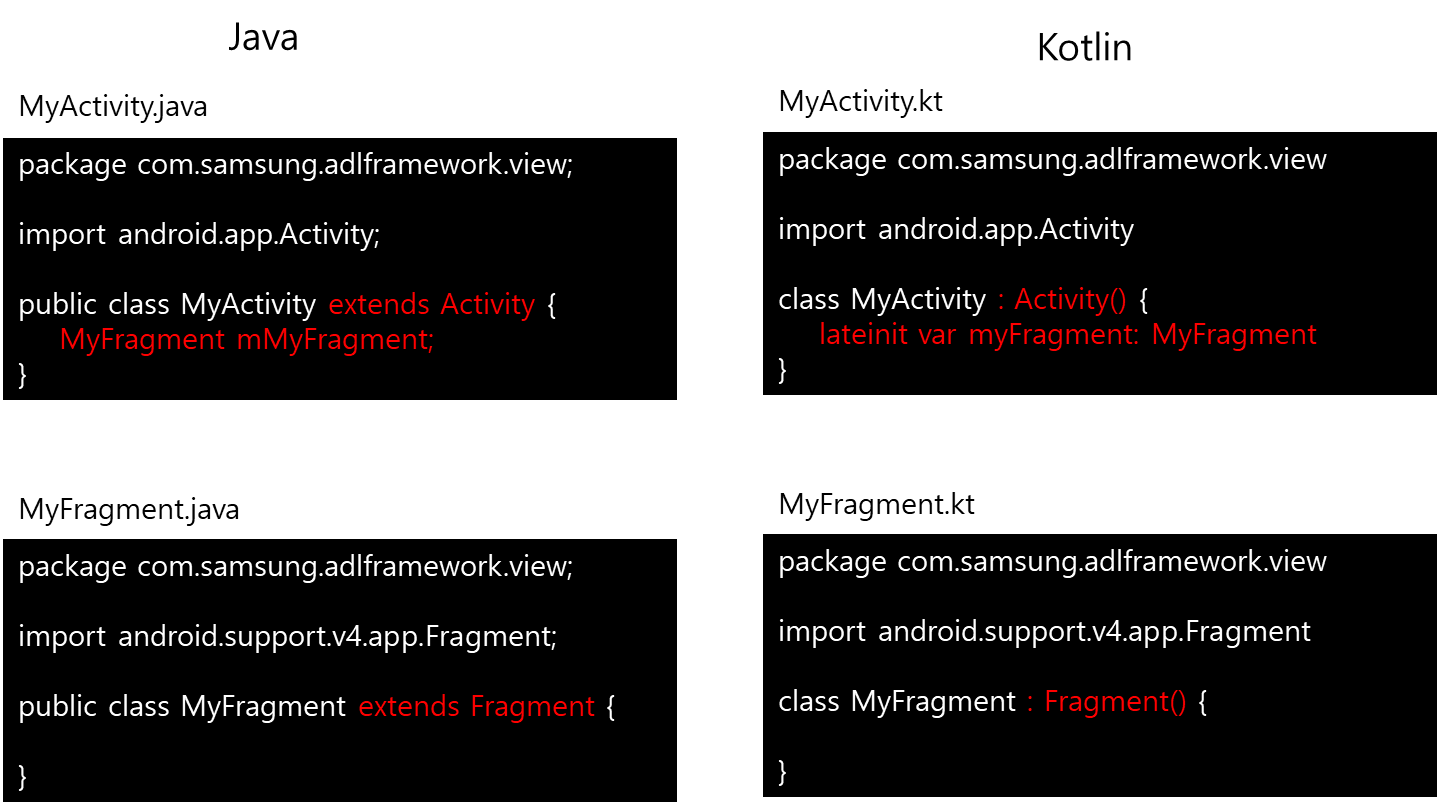


그림 . Code Generate: Java vs Kotlin

* 1. NFR\_01. ADL Parsing 중 Error 발생시 회복 시간

ADL Parsing중에는 유효하지 않은 Component Type을 기입하거나, 필수적인 Attribute가 존재하지 않는 등 ADL에서 정의한 Element, Relation 기술 방법에 맞지 않게 ADL Code가 작성된 경우 Error를 발생시키고 다시 Translate 요청을 받을 수 있는 상태로 전환되야 한다. 이를 위한 후보구조를 검토한다.

* + 1. 후보구조 . ADL 분석 단계에서 Error 발생시 Thread 종료

ADL 분석 단계에서 Error가 발생하면 전체 Translate 동작을 종료할 필요가 있다. 이에 어휘 분석 및 구문 분석, 의미 분석 구간에서 Error 발생 시 Error Handler에게 이를 알리고 TaskManager는 Error Handler를 통해 Error 상황을 인지한다. 이후 TaskManager는 다시 요청을 받을 수 있는 Ready 상태로 State를 변경하고, Translate Task를 Destroy한다. Task Destroy시 내부 Thread를 모두 종료한다.



그림 . Error Handler

* + 1. 후보구조 . Translate 동작의 Multitasking

항상 Translate Request를 받을 수 있도록 한다. Translate 요청 시 Translate Task를 별도 Thread로 Execute 시키고 바로 다시 Translate Request를 받을 수 있도록 설계한다. Translate Task는 동작 완료 시 Graph나 Error를 전달하게 되며 해당 결과에 관계없이, 즉 별도의 회복시간 없이 사용자는 언제든 다시 Translate 요청을 내릴 수 있다.



그림 . Translate 동작의 Multitasking

1. 후보 구조 평가
   1. QA\_01 ADL Code 작성의 편의성에 관한 후보구조 평가

**후보구조 1. Element - Relation으로만 구성된 ADL**(채택함)

**후보구조 2. 기존 Markup language의 문법 사용(채택하지 않음)**

**후보구조 3. Programming Language형태의 ADL 명세(채택하지 않음)**

|  |  |  |  |
| --- | --- | --- | --- |
| 품질 요구사항 | 후보구조 1 | 후보구조 2 | 후보구조 3 |
| QA\_01 ADL Code 작성의 편의성 | (++) 간결하게 ADL Code 작성할 수 있다. | (+) 따로 문법을 학습할 필요가 없다. | (+) Programming하듯이 작성이 가능하다. |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (-) 간결한 Code여서 Group 등의 문법사항을 고려해야 하며 Parsing에 다소 고려사항이 많다. | (+) Parsing하기 수월한 Code이다. | (--) Code량이 많고 각 method의 의미 분석이 필요해서 성능에 불리하다. |
| QA\_04 Element Type 변경 용이성 | (+) Element Type을 그대로 사용하므로 변경에 대한 영향이 없다. | (+) Element Type을 그대로 사용하므로 변경에 대한 영향이 없다. | (-) Element Type에 따라 Method가 추가 / 삭제 되어야 한다. |
| QA\_06 Autocomplete 완료시간 | (+) Keyword list를 추가 작업이 없이 그대로 참조하기 때문에 Search가 빠르다. | (+) Keyword list를 추가 작업이 없이 그대로 참조하기 때문에 Search가 빠르다. | (-) Method List를 Element type에 따라 따로 관리해야 해서 Search가 상대적으로 느리다. |

후보구조 1은 본 시스템의 차별화된 사용성을 만들 수 있는 구조로 채택한다. 해당 후보구조 설명에 사용된 component가 7개 Relation이 3개 있는 구조를 간결성 성능 측정시에도(낮을수록 좋음) 더 우수하게 측정된다.

* **후보구조 1: 0.69[간결성] = 599[ADL Code 문자 수] / 867[변환된 DOT 코드 문자 수]**
* 후보구조 2: 1.07[간결성] = 936[ADL Code 문자 수] / 867[변환된 DOT 코드 문자 수]
* 후보구조 3: 0.96[간결성] = 840[ADL Code 문자 수] / 867[변환된 DOT 코드 문자 수]

후보구조 1은 간결성이 매우 우수하며, Element와 Relation으로만 구성되어 있어서 새로운 ADL 언어 학습에도 어려움이 없을 것으로 판단한다. 또한 ‘후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계를 지원하기 위해서는 Line 별로 Stream으로 발행되어 분석이 가능해야 하는데, 이에 적합한 구조로 볼 수 있다. 하지만 후보구조2에 비해 Parsing이 좀 더 복잡할 수 있는 Risk가 존재한다. 이는 작성 편의성을 위한 Tradeoff로 기계가 인지하기 쉬운 Markup language보다는 사람이 인지하기 쉬운 후보구조 1의 ADL이 보다 작성에 있어서 편의성을 제공한다. 우선순위에 따라 후보구조 1을 채택하며 Parsing작업도 Markup language대비 Grouping등에서 조금 더 고려할 부분이 있을 뿐 programming Language 형태 대비해서는 월등히 빠른 분석이 가능하다.

**후보구조 4. 확장 가능한 Component Type(채택함)**

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (+) 반복된 Element나 Relation의 정의를 간략하게 표현할 수 있다. |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (-) 확장된 Component Type을 비교하며 Syntax 분석 단계에서 치환을 해줘야 해서 다소 성능상에 악영향을 끼친다. |
| QA\_06 Autocomplete 완료시간 | (-) Autocomplete를 위한 prefix 비교 시 확장된 Component Type도 확인이 필요하며 이에 따라 다소 성능상에 악영향을 끼친다. |

후보구조 4는 ADL 작성의 편의성을 높여주는 구조로 Graph Description Language (GDL) 변환시간이나 Autocomplete 완료시간에 다소 악영향을 끼치지만 ADL 작성의 편의성을 위한 Tradeoff로 우선순위에 따라 채택한다. QA\_02와 QA\_06의 성능저하 Risk는 그 영향이 미미해서 편의성을 선택하는 것이 더 효과적이다. GDL 변환시간에는 선언된 Component Type을 Syntax Analyzer에서 확인 후 선언 시 사용된 Element Type 및 Attribute Type을 적용하게 되는데, 확인하는 시간에서 다소간의 시간상 손해가 있을 뿐 이후 Element Type 및 Attribute Type을 분석할 필요가 없이 그대로 copy하게 됨에 따라 상황에 따라 후보구조 4를 적용하지 않고 Element Type 및 Attribute Type을 작성한 것 보다 성능상에 이점이 있을 수 있다. 또한 Autocomplete 완성시간도 후보구조 18. Radix Tree로 Keyword 관리 구조 적용시 Tree의 항목이 추가되는 정도이며 그 영향이 크지 않다.

* 1. QA\_02 ADL Code의 Graph Description Language 변환성능에 관련한 후보구조 평가

**후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계** (채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (+) Line 별 Stream으로 발행 시 병렬적으로 처리가 가능함에 따라 후보구조1과 같은 보다 간략한 ADL Code 작성의 이점을 높일 수 있다. |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (++) Code의 Line 별로 각 Filter가 병렬 처리됨에 따라 변환시간이 빠르다. |
| QA\_03 Translate 요청 시 첫 Graph 반환 시간 | (+) Graph Description Language로 변환할 수 있는 Parse Tree가 빨리 완성됨에 따라 Graph 반환시간이 빨라진다. |

후보구조 5는 Lexical Analysis – Syntax Analysis – Semantic Analysis로 이어지는 Parsing 단계를 병렬로 처리할 수 있도록 ADL Code를 구분하며, 이에 따라 Graph Description Language Code 변환 및 첫 Graph 반환시간에도 이점이 있어서 채택한다. Line 별로 해석이 가능한 후보구조 1과 밀접한 연관성이 있으며 본 구조가 후보구조 1의 장점을 극대화할 수 있다.

**후보구조 6. Update를 위해 Token 및 Parse Tree 유지** (채택하지 않음)

Translate한 이후 다시한번 Translate시 Parsing 속도를 높이기 위한 방식으로, ADL Code의 추가 / 변경시에는 좋은 성능을 발휘 할 수 있으나, Line 기반으로 Parse Tree를 유지해야 해서 삭제시에는 전체 Parse Tree를 update해야해서 오히려 실행 속도가 느려질 수 있는 RISK가 존재한다. 또한 Memory상에 Token들과 Parse Tree를 항상 유지해야 해서 Memory 사용측면에서도 문제가 발생할 수 있다. 작은 변경에는 이점이 있을 수 있으나, 이점대비 Logic의 복잡성과 오류의 위험이 높아 채택하지 않는다.

**후보구조 7. Token, Tree 구성을 위한 Resource 선 할당** (채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (+) Token 및 Tree 구성을 위한 Memory할당 / 해지의 load를 회피할 수 있다. |
| QA\_03 Translate 요청 시 첫 Graph 반환 시간 | (+) Token 및 Tree 구성에 시간 소요가 감소함에 따라 첫 Graph 반환시간에도 플러스 요인이 된다. |
| Memory Efficiency | (-) 사용여부와 관계없이 Memory를 점유하고 있어서 전체 System의 Memory 효율을 낮춘다. |

후보구조 7은 일반적인 View 구성을 위한 Token과 Tree를 Memory상에 미리 적재해 둠으로써 Translate시에 Memory 할당 / 해제에 대한 load를 줄이는 구조이며, 이로 인한 성능상의 이점이 있어서 채택한다. 하나의 View를 구성하는 ADL Code는 그 Size가 제한적이며, 이에 대한 Memory 점유가 시스템에 끼치는 영향은 미미할 것으로 판단되며, Memory 효율성에 대한 Risk는 후보구조 8을 통해 일정 부분 해소한다.

**후보구조 8. Tree 구성을 위한 Resource 효율화** (채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (+) Tree 생성시 Token을 그대로 사용하기 때문에 Tree 생성시 Memory 할당에 대한 load가 줄어든다.. |
| QA\_03 Translate 요청 시 첫 Graph 반환 시간 | (+) Tree 생성이 빨라 짐에 따라 Graph 반환 성능에도 긍정적인 영향을 끼친다. |
| Memory Efficiency | (+) Lexical Analysis에서 생성한 Token을 Syntax Tree 및 Parse Tree에서 그대로 사용하기 때문에 memory 효율이 높아진다. |

후보구조 8은 ADL Parsing Pipe & Filter 과정 속에서 Token을 공유 데이터 영역 Repository로 구성하고 이를 활용해서 Tree를 구성한다. Tree Node의 Value를 별도로 생성하지 않아 Memory 효율을 높일 수 있다. 이에 따라 Memory 할당에 대한 load를 줄일 수 있고 이는 성능향상에 도움이 되는 구조로 채택한다.

**후보구조 9. Dot language Parser library를 사용** (채택하지 않음)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (-) Python Programming language 형태로 작성이 되어야 하며, 이는 View를 그리는데 직관적이지 않다. |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (+) Open Source Library로 사용성이 검증되어 있다. 이에 따라 변환 성능은 높을 것으로 판단된다. |
| QA\_04 Element Type 변경 용이성 | (-) 고정된 Library상의 API가 존재하며 Element Type 변경 시 이를 적용하기가 어렵다. |
| QA\_05 Graph Visualization Library 변경 용이성 | (-) DOT으로만의 변경을 지원하기 때문에 Library 변경에 취약하다. |
| Parser 개발비용 | (++) Open Source Library를 사용함에 따라 별도의 개발비용이 발생하지 않음 |

검증된 Open Source Library를 통해 DOT Language로 변환하는 구조로써, Parsing에 대한 개발 비용을 줄일 수 있다. 하지만 DOT이외의 Graph Description Language로의 확장성이 떨어지고, ADL또한 Python으로 작성이 되어야 하는 문제와, Element Type 변경에 제약이 많고 Code Generation시에 사용하는 Parse Tree도 사용할 수 없어서 추후 변경용이성에 제약이 많아 채택하지 않는다.

* 1. QA\_03 Translate 요청 시 첫 Graph 반환 성능 관련 후보구조 평가

**후보구조 10. 연속적으로 Graph를 갱신** (채택하지 않음)

**후보구조 11. 일정한 주기로 Graph를 갱신** (채택함)

**후보구조 12. Translate 요청전에 Background parsing** (채택하지 않음)

|  |  |  |  |
| --- | --- | --- | --- |
| 품질 요구사항 | 후보구조 10 | 후보구조 11 | 후보구조 12 |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (-) 지속적인 Graph 반환 동작으로 GDL 변경 성능에 저하를 일으킬 수 있음 | (0) GDL 변환 전에 Graph 반환이 일어나지만 반복된 동작이 아니어서 끼치는 영향은 무시할 수 있는 수준으로 판단됨 | (0) 변화시간 자체에는 영향이 없다. |
| QA\_03 Translate 요청 시 첫 Graph 반환시간 | (+) 한 line이라도 parsing이 되면 Graph를 생성하므로 빠른 성능을 보인다. | (+) 한 line이라도 parsing이 되면 Graph를 생성하므로 빠른 성능을 보인다. | (+) background에서 parsing 된 정보를 바로 graph로 변환해서 반환시간이 빠르나, Code 변경 직후 Translate 요청 시 첫 Graph 반환 시간은 parsing이 모두 완료되고 진행되야 해서 느릴 수 있다. |
| Process Workload | (-) 지속적인 Graph 변환으로 Translate시 Process가 Work Load가 크다. | (0) 기본 Translate 동작과 큰 차이가 없다. | (--) Code 변경시마다 background parsing을 진행하게 돼서, Process의 동작 점유가 크다. |

Translate시 즉각적인 Graph 반환 성능면에서 세가지 후보구조가 모두 비슷한 성능을 보이나, 후보구조 10은 각 Token마다 Graph를 갱신함에 따라 첫 Graph 반환 성능은 높겠으나, 너무 빈번한 Graph 생성으로 System Resource 점유가 높아져 GDL 변환 성능에 악영향을 끼치며 이에 따라 최종 Graph 생성에도 지장이 발생할 수 있어서 채택하지 않는다. 후보구조 12는 상황에 따라서는 가장 빠른 Graph 변환 동작이 이루어질 수 있으나, ADL 도구에서 Background Translate 요청을 지속적으로 요청해야 하는 부분과, 이로 인한 프로세스의 System Resource 점유 문제로 채택하지 않는다. 후보구조 11은 후보구조 10. 연속적으로 Graph를 갱신의 장점인 첫 Graph를 즉각적으로 반환하는 성능과 동일한 성능을 발휘하고 RISK였던 빈번한 Graph 생성에 따른 Resource 점유 문제도 일정시간 주기를 두고 진행함에 따라 GDL 변환 성능에 끼치는 영향을 최소화했다. 이에 따라 최종 Graph 생성에도 큰 지연을 발생시키지 않는 구조로 평가되며 이에 따라 채택한다.

* 1. QA\_04 Element Type 변경 용이성 관련 후보구조 평가

**후보구조 13. 별도의 AppModel library로 지연 바인딩**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (+) Element Type의 추가로 ADL Code 작성이 더 수월해짐 |
| QA\_04 Element Type의 변경 용이성 | (++) ADL Framework의 어떠한 변경도 없이 Library 변경만으로 Element Type 변경 가능 |

Android의 다양한 Element들을 ADL 언어차원에서 지정할 수 있도록 정의한 Element Type을 AppModel library 교체만으로 변경할 수 있게 함으로써 Element Type의 변경 용이성을 극대화할 수 있다고 평가된다. Element Type을 변경하고자 할 때 ADL Framework 자체를 빌드 / 테스트 / 배포할 필요없이 Element Type에 대한 AppModel Library만 빌드 / 테스트 / 배포하면 됨에 따라 개발 및 발행의 효율성이 증대된다.

**후보구조 14. Element Type에 Decorate** Pattern을 적용(채택하지 않음)  
**후보구조 15. Element Type에 Strategy** Pattern을 적용(채택함)

|  |  |  |
| --- | --- | --- |
| 품질 요구사항 | 후보구조 14 | 후보구조 15 |
| QA\_04 Element Type 변경 용이성 | (+) Element Type을 추가하고자 할 때 유용하다. | (++) 다양한 형태의 Element Type을 상황에 맞게 변경하며 사용할 수 있다. |

두가지 구조 모두 손쉽게 Element Type을 변경할 수 있는 구조이며, 변경 시 기존 모듈의 영향이 적어서 변경 용이성이 높다고 볼 수 있다. 하지만 후보구조 13의 형태로 AppModel을 별도 Library로 배포할 때 Android 뿐 아니라, iOS, Windows등의 다양한 platform을 위한 Element Type을 지원하도록 확장될 수 있는데, 이를 위해서는 각 설정에 따라 AppModel을 변경하며 사용하는 후보구조 15. Strategy Pattern 형태로 적용되는 것이 적합하다. 후보구조 14는 AppModel 변경이 아닌 AppModel을 확장하며 사용하는 경우에 용이한 구조로 현재 사용 목적에 부합하지 않아서 채택하지 않는다.

* 1. QA\_05 Graph Visualization Library 변경 용이성 관련 후보구조 평가

**후보구조 16. 별도의 GraphDrawer Library로 지연 바인딩(**채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_05 Graph Visualization Library 변경 용이성 | (++) 최소한의 ADL Framework 변경과 GraphDrawer library 변경만으로 Graph Visualization Library 변경 가능 |

Graph Visualization Library를 포함하는 형태의 GraphDrawer library는 ADL Framework과 동일한 interface로 동작할 수 있게 하는 Adapter로써, IGraphDrawer의 구현체이다. GraphDrawer를 별도의 Library로 발행하고 이를 변경하는 방식으로 Graph Visualization Library 변경 용이성을 높일 수 있다. Graph를 생성하는 Visualization Library를 변경하고자 할 때 ADL Framework 자체를 빌드 / 테스트 / 배포할 필요없이 GraphDrawer Library만 빌드 / 테스트 / 배포하면 됨에 따라 개발 및 발행의 효율성이 증대된다. 이에 따라 Graph Drawer Library를 생성하고 이를 지연 바인딩하는 후보구조 16을 채택한다.

**후보구조 17. GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_05 Graph Visualization Library 변경 용이성 | (++) 최소한의 ADL Framework 변경과 Drawer Library, GraphDrawer Library 변경으로 다양한 조합의 Graph 변환이 가능 |

후보구조 17은 다양한 Graph Description Language(GDL) 변환 동작을 IDrawer로 추상화하고 이에 대한 구현체를 별도 Library로 발생한다. 그리고 GDL을 Graph로 표현하는 다양한 Graph Visualization Library를 IGraphDrawer로 추상화하고 마찬가지로 구현체를 별도 Library로 발행한다. Bridge Pattern을 적용함으로써 각각의 조합이 가능하고 Parse Tree의 Graph 변환은 Graph Description Language와 Graph Visualization library와 관계없이 동일한 Interface로 동작하게 된다. 변경 용이성을 높일 수 있는 구조로 후보구조 17을 채택한다.

* 1. QA\_06 Autocomplete 완료 성능 관련 후보구조 평가

**후보구조 18. Radix Tree로 Keyword 관리**(채택함)

**후보구조 19. 문맥에 따른 Autocomplete 적용**(채택함)

**후보구조 20. ADL 도구에 Keyword Array 전달**(채택하지 않음)

|  |  |  |  |
| --- | --- | --- | --- |
| 품질 요구사항 | 후보구조 18 | 후보구조 19 | 후보구조 20 |
| QA\_01 ADL Code 작성의 편의성 | (+) 빠른 Autocomplete 제공으로 작성이 편의성 향상 | (++) 문맥에 맞는 Autocomplete Guide 제공으로 작성의 편의성 향상 | (0) ADL 도구에서 Keyword Array를 어떻게 활용하는지에 따라 다름 |
| QA\_06 Autocomplete 완료시간 | (++) Radix Tree를 통해 Prefix 비교 시 최소한의 연산으로 Keyword 추출이 가능 | (+) 문맥에 따른 별도 list를 확인해서 적은 연산으로 keyword 추출이 가능 | (+) 별도의 검색없이 전체 List를 반환하기에 응답이 빠름 |
| ADL 도구 구현 용이성 | (+) ADL Framework의 API 호출만으로 손쉽게 구현 가능 | (+) ADL Framework의 API 호출만으로 손쉽게 구현 가능 | (-) ADL 도구내에서 prefix 검색을 위한 추가 Logic 구현이 필요함 |

후보구조 18은 ADL Framework 구동 시 AppModel에 따른 Element Type, Element Attribute, Relation Type, Relation Attribute를 list up하고 이에 대한 Radix Tree 생성한다. 최소한의 Prefix 문구 비교연산으로 Keyword list 추출이 가능하며 이에 따라 채택한다.

후보구조 19 은 ADL 작성 문맥상 Attribute 기입을 위한 괄호가 열렸는지에 따라 Element Type을 추천해야 하는 것인지, Element Attribute를 추천해야 하는 것인지, Relation Attribute를 추천해야 하는지를 확인한다. 그리고 이에 맞는 Autocomplete keyword list를 반환한다. 보다 높은 사용편의성을 제공해주며 성능 또한 제한된 List에서 검색하게 되므로 향상될 수 있고 이에 따라 채택한다.

후보구조 20 은 전체 keyword list를 ADL 도구에 전달하고 ADL 도구에서 Autocomplete 기능을 구현하는 방식으로, EditText를 가지고 있는 ADL 도구가 보다 빠르게 Autocomplete 추천이 가능 할 수 있으나, ADL 도구 개발자의 구현 부담이 상당히 높아서 채택하지 않는다.

* 1. QA\_07 ADL Framework API의 확장성 관련 후보구조 평가

**후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용**(채택함)

**후보구조 22. Task 기반의 Event Bus 적용**(채택하지 않음)

|  |  |  |
| --- | --- | --- |
| 품질 요구사항 | 후보구조 21 | 후보구조 22 |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (+) Direct call로 인해 Translate시작이 즉각적이고 빠르다. | (-) Event Bus를 통한 Communication으로 상대적으로 지연이 발생한다. |
| QA\_03 Translate 요청 시 첫 Graph 반환 시간 | (+) Direct call로 인해 Translate시작이 즉각적이고 빠르며, 반환되는 Graph도 Subscriber에 바로 전달된다. | (-) Event Bus를 통한 Communication으로 상대적으로 지연이 발생한다. |
| QA\_04 Element Type 변경 용이성 | (-) AppModel을 변경하는 API를 추가하고자 할 때 상대적으로 Core layer와 상호작용을 위한 구현이 많다. | (+) AppModel을 변경하는 API를 추가하고자 할 때 event bus로 event를 전달하기만 하면 돼서 상대적으로 간단하게 구현이 가능하다. |
| QA\_05 Graph Visualization Library 변경 용이성 | (-) Drawer와 GraphDrawer를 변경하는 API를 추가하고자 할 때 상대적으로 Core layer와 상호작용을 위한 구현이 많다. | (+) Drawer와 GraphDrawer를 변경하는 API를 추가하고자 할 때 event bus로 event를 전달하기만 하면 돼서 상대적으로 간단하게 구현이 가능하다. |
| QA\_07 ADL Framework API의 확장성 | (+) Core Layer와 독립적으로 개발이 가능하며 기능에 따라 publish – subscriber만 추가해 주면 됨 | (++) ADL Framework의 API 호출만으로 손쉽게 구현 가능 |
| QA\_08 Code Generate Language 확장성 | (-) Code Generate Language를 변경하는 API를 추가하고자 할 때 상대적으로 Core layer와 상호작용을 위한 구현이 많다. | (+) Code Generate Language를 변경하는 API를 추가하고자 할 때 event bus로 event를 전달하기만 하면 돼서 상대적으로 간단하게 구현이 가능하다. |

후보구조 21과 후보구조 22 모두 ADL 도구와 Communication하는 모듈을 Interface layer로 구분하고 Core logic을 Core Layer로 배치한다. 후보구조 21은 새로운 API가 추가될 때마다 Publish – Subscriber 패턴에 맞게 각각을 추상화하고 구현체를 생성해야 한다. 하지만 후보구조 21 은 별도의 path를 만들 필요없이 Event bus를 통해 Communication이 가능하다. 이에 따라 변경 용이성 측면에서는 후보구조 22이 더 나은 구조라 할 수 있겠으나, Event Bus 방식은 Direct call이아니여서, 성능에 악영향을 끼친다. 성능과 변경 용이성의 Tradeoff 관계에 있는 상황으로 우선순위에 따라 후보구조 21을 채택한다. 후보구조 21은 후보구조 22에 비해 상대적으로 변경 용이성이 떨어지는 것이지 그 자체로는 변경 용이성이 높다고 할 수 있다. 변경이 잦을 수 있는 Interface Layer가 Core Layer로만 Dependency가 있어서 독립적인 개발이 용이하다.

* 1. QA\_08 Code 생성 Language 확장성 관련 후보구조 평가

**후보구조 23. Code Generation Library 적용**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_08 Code Generate Language 확장성 | (++) ADL Framework의 변경없이 Library 변경만으로 Code Generate Language 변경 가능 |

후보구조 13. 별도의 AppModel library로 지연 바인딩처럼 별도의 Library로 Code 생성을 위임하는 형태여서 변경 용이성이 높다고 할 수 있다. Code Generation을 위한 Language 변경 시 ADL Framework 자체를 빌드 / 테스트 / 배포할 필요없이 Code Generation에 대한 Library만 빌드 / 테스트 / 배포하면 됨에 따라 개발 및 발행의 효율성이 증대된다. 이에 따라 채택한다.

**후보구조 24. Language별 Class Prototype변경**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_08 Code 생성 Language 확장성 | (+) 다른 모듈에 영향이 거의 없게 변경이 가능 |

사용자가 필요에 따라 Language 설정을 변경해서 Code Generate를 하는 사용성에 부합하는 전략은 Strategy Pattern 형태로 Class의 Prototype을 설정하는 것이다. 추가나 변경이 필요한 경우에 관련 Module만 변경하면 가능해서 변경 용이성이 높다고 평가하여 채택한다.

* 1. NFR\_01 ADL Parsing중 Error 발생시 회복 시간 후보 구조 검토

**후보구조 25. ADL 분석 단계에서 Error 발생시 Thread 종료**(채택함)

**후보구조 26. Translate 동작의 Multitasking**(채택하지 않음)

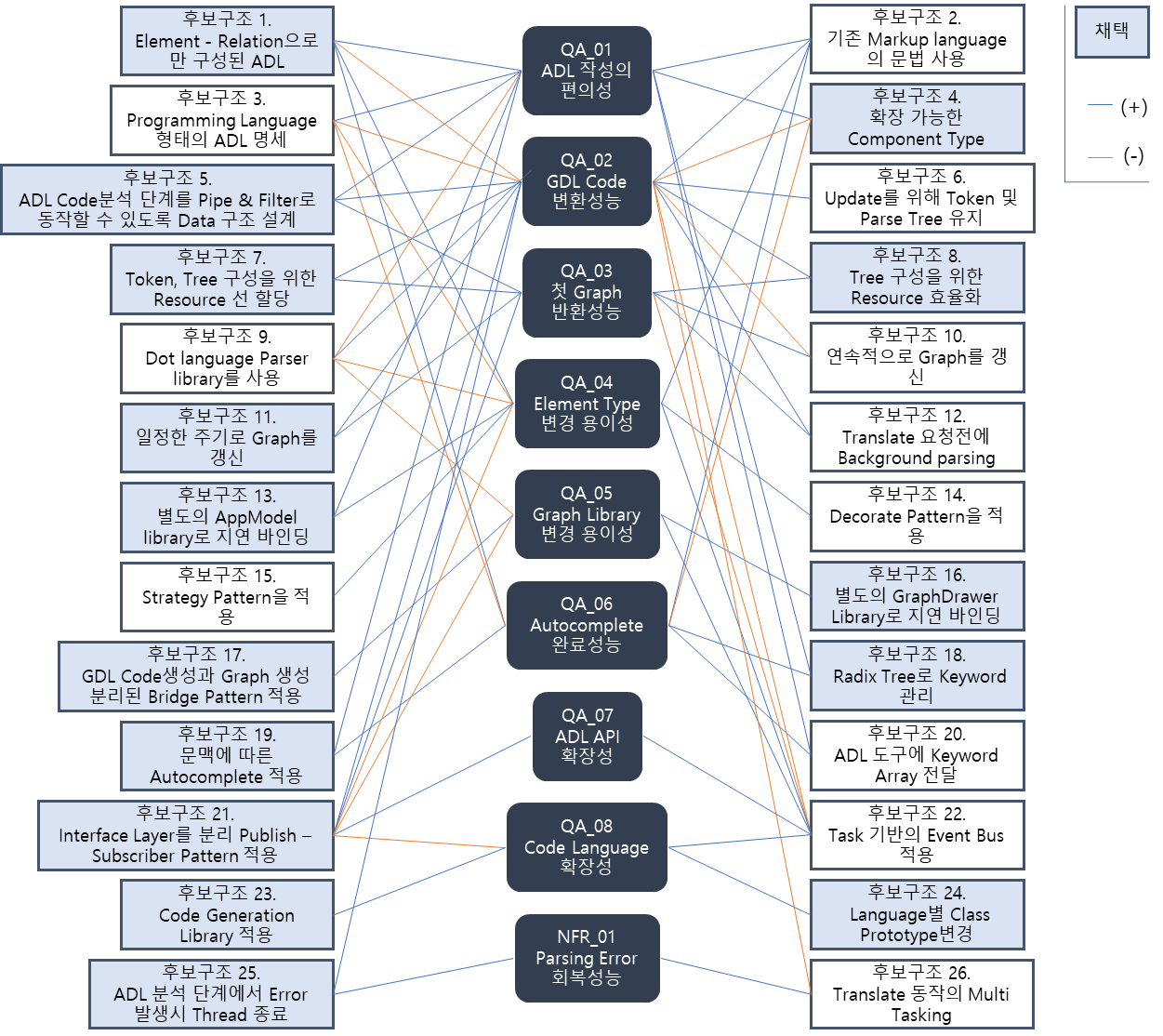
|  |  |  |
| --- | --- | --- |
| 품질 요구사항 | 후보구조 25 | 후보구조 26 |
| QA\_02 ADL Code의 Graph Description Language Code 변환 시간 | (+) Error 발생시 불필요한 변환과정이 바로 종료됨에 따라 Graph Description Language Code 변환작업을 바로 마치게 됨 | (-) Error가 발생해도 변환과정은 계속 이루어지고 일정 시간 후에 변환작업이 종료된다. |
| NFR\_01 ADL Parsing중 Error 발생시 회복 시간 | (+) Error 발생시 바로 Task를 Destroy하고 State를 변경해서 회복이 빠름 | (++) Error 발생과 관계없이 항상 Translate 요청을 받을 수 있는 상태이다. |

후보구조 25 는 Translate 과정에서 ADL 문법에 맞지 않는 등의 Error 발생시 Error Handler가 이를 인지하고 TaskManager를 통해 Task를 바로 종료할 수 있어서 Error 발생에 대한 회복 시간이 제약조건인 1000ms 이하로 이루어질 것으로 판단되며 후보구조 26은 이러한 회복 시간도 필요없이 항상 Translate 요청을 받을 수 있는 구조이다. NFR\_01은 일정시간내로 회복이 되기만 하면 되는 품질 요구사항으로 두 후보구조 모두 채택가능한 구조이나 불필요한 변환 과정이 Error발생시에도 지속되는 점에 따라 System 성능에 악영향을 끼칠 수 있어서 후보구조 25를 채택한다.

* 1. 후보 구조 평가 결과

|  |  |
| --- | --- |
| 후보구조 | 채택 여부 |
| 후보구조 1. Element - Relation으로만 구성된 ADL | 채택함 |
| 후보구조 2. 기존 Markup language의 문법 사용 | 채택하지 않음 |
| 후보구조 3. Programming Language형태의 ADL 명세 | 채택하지 않음 |
| 후보구조 4. 확장 가능한 Component Type | 채택함 |
| 후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계 | 채택함 |
| 후보구조 6. Update를 위해 Token 및 Parse Tree 유지 | 채택하지 않음 |
| 후보구조 7. Token, Tree 구성을 위한 Resource 선 할당 | 채택함 |
| 후보구조 8. Tree 구성을 위한 Resource 효율화 | 채택함 |
| 후보구조 9. Dot language Parser library를 사용 | 채택하지 않음 |
| 후보구조 10. 연속적으로 Graph를 갱신 | 채택하지 않음 |
| 후보구조 11. 일정한 주기로 Graph를 갱신 | 채택함 |
| 후보구조 12. Translate 요청전에 Background parsing | 채택하지 않음 |
| 후보구조 13. 별도의 AppModel library로 지연 바인딩 | 채택함 |
| 후보구조 14. Element Type에 Decorate Pattern을 적용 | 채택하지 않음 |
| 후보구조 15. Element Type에 Strategy Pattern을 적용 | 채택하지 않음 |
| 후보구조 16. 별도의 GraphDrawer Library로 지연 바인딩 | 채택함 |
| 후보구조 17. GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용 | 채택함 |
| 후보구조 18. Radix Tree로 Keyword 관리 | 채택함 |
| 후보구조 19. 문맥에 따른 Autocomplete 적용 | 채택함 |
| 후보구조 20. ADL 도구에 Keyword Array 전달 | 채택하지 않음 |
| 후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용 | 채택함 |
| 후보구조 22. Task 기반의 Event Bus 적용 | 채택하지 않음 |
| 후보구조 23. Code Generation Library 적용 | 채택함 |
| 후보구조 24. Language별 Class Prototype변경 | 채택함 |
| 후보구조 25. ADL 분석 단계에서 Error 발생시 Thread 종료 | 채택함 |
| 후보구조 26. Translate 동작의 Multitasking | 채택하지 않음 |

각 후보구조의 품질 속성에 대한 전체 영향평가는 아래와 같다.



1. 최종 구조 설계
   1. 시스템의 변경용이성 측면

아래는 변경 용이성 측면에 의해 Layered Architecture로 설계된 본 시스템의 최종구조 Module View이다.



그림 . 최종구조 Module View

|  |
| --- |
| 후보구조 13. 별도의 AppModel library로 지연 바인딩  후보구조 15. Element Type에 Strategy Pattern을 적용  후보구조 16. 별도의 GraphDrawer Library로 지연 바인딩  후보구조 17. GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용  후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용  후보구조 23. Code Generation Library 적용  후보구조 24. Language별 Class Prototype변경 |

상기 후보구조에 따라 시스템을 크게 3가지 Layer로 구분하여 각 Layer별 독립적인 개발이 가능하고 변경이 용이하도록 설계하였다. ADL Tool과의 Interface를 담당하는 Interface Layer와 실제 Translate 및 Code Generate, Autocomplete기능을 담당하는 Core Layer, 변경이 자주 발생할 수 있어서 별도 Library로 발행되는 Extension Layer가 있다.

* + 1. Extension Layer – Core Layer
* 후보구조 13. 별도의 AppModel library로 지연 바인딩
* 후보구조 15. Element Type에 Strategy Pattern을 적용
* 후보구조 16. 별도의 GraphDrawer Library로 지연 바인딩
* 후보구조 17. GDL Code생성과 Graph 생성 분리된 Bridge Pattern 적용
* 후보구조 23. Code Generation Library 적용
* 후보구조 24. Language별 Class Prototype변경

상기 후보구조들에 의해 변경이 발생할 수 있는 부분들을 Extension Layer로 분리하고 별도 Library에서 동작하도록 설계한다. Core Layer에서 정의한 Interface의 Concrete Module을 각각의 Library에서 구현하며 각 Library들은 Strategy Pattern 및 Bridge Pattern을 통해 상황에 따라 변경 조합하여 사용할 수 있도록 구성하였다.



그림 . Extension Layer – Core Layer Component & Connector View

* + 1. Interface Layer – Core Layer
* 후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용

상기 후보구조에 따라 Interface Layer와 Core Layer로 구분하고 각 Layer의 독립적인 개발을 위해 Acyclic Dependencies Principle에 따라 Publish – Subscriber Pattern으로 설계하였다.

ADL 도구에 ADL Framework API를 제공하는 ADL Manager와 ADL의 Event를 ADL 도구에 전달하기 위한 ADL Listener interface를 정의하였다. 마찬가지로 Core Layer에서는 각 Task 실행을 위한 API를 제공하는 Task Manager와 Core Layer로부터 Event를 Interface Layer에 전달하기 위한 Task Observer interface를 정의하였다. Interface Layer에서는 해당 Interface에 대한 Concrete Module을 가지고 있다.



그림 . Interface Layer – Core Layer Component & Connector View

* 1. 시스템의 성능 측면

아래는 성능측면이 고려된 Process - Thread관점에서의 본 시스템의 최종구조 Deployment View이다.



그림 . 최종구조 Deployment View

앞서 설명한 변경용이성에 따라 3-Tier로 구성되어 있으며,

|  |
| --- |
| 후보구조 1. Element - Relation으로만 구성된 ADL  후보구조 4. 확장 가능한 Component Type  후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계  후보구조 7. Token, Tree 구성을 위한 Resource 선 할당  후보구조 8. Tree 구성을 위한 Resource 효율화  후보구조 11. 일정한 주기로 Graph를 갱신  후보구조 18. Radix Tree로 Keyword 관리  후보구조 19. 문맥에 따른 Autocomplete 적용 |

상기 후보구조를 고려해서 효율적인 Translate를 위해 병렬 Thread로 동작할 수 있도록 Pipe & Filter Architecture를 적용하였고, 해당 구조의 우수한 변경용이성 특성에 따라 Filter를 교체를 통해 문맥에 맞는 Autocomplete 동작을 할 수 있도록 설계하였다.

* + 1. Translate 성능
* 후보구조 1. Element - Relation으로만 구성된 ADL
* 후보구조 4. 확장 가능한 Component Type
* 후보구조 5. ADL Code분석 단계를 Pipe & Filter로 동작할 수 있도록 Data 구조 설계
* 후보구조 7. Token, Tree 구성을 위한 Resource 선 할당
* 후보구조 8. Tree 구성을 위한 Resource 효율화
* 후보구조 11. 일정한 주기로 Graph를 갱신

상기 후보구조에 따라 ADL 전체가 Parsing 되지 않은 상태에서도 한 Line 만으로도 의미를 가진 Parse Tree를 생성할 수 있고 이에 따라 각 Line을 Pipe & Filter 병렬 쓰레드 방식으로 어휘 분석, 구문 분석, 의미 분석 및 Graph 생성 진행한다. 또한 일정 주기로 Graph를 갱신할 수 있도록 Scheduled Queue를 두고 Graph 변환을 시도하며, ADL Framework 구동 시 Token Repository를 미리 할당하고 해당 Token Repository를 Syntax Tree 및 Parse Tree가 참조하도록 설계하여 메모리 할당에 대한 Load를 줄이고 메모리 효율을 높였다.



그림 . Translate Component & Connector View

* + 1. Autocomplete 성능
* 후보구조 18. Radix Tree로 Keyword 관리
* 후보구조 19. 문맥에 따른 Autocomplete 적용

상기 구조에 따라 Autocomplete Task가 AppModel을 참조해서 별도의 Radix Tree를 관리하게 되고, Translate를 위한 Pipe & Filter 구조에서 Syntax Analyzer Filter를 Autocomplete를 위한 Autocomplete Syntax Analyzer Filter로 교체해서 문맥에 맞는 추천을 할 수 있게 설계했다.



그림 . Autocomplete Component & Connector View

* 1. 시스템의 신뢰성 측면

|  |
| --- |
| 후보구조 25. ADL 분석 단계에서 Error 발생시 Thread 종료 |

ADL 분석 단계에서 Error 발생시 진행중이던 Translate 작업을 즉각적으로 종료하고 다시 Translate 요청을 받을 수 있는 상태로 진입한다. 본 시스템은 이를 위해 Error Handler를 배치하고 각 변환작업에서 Error발생시 Error Handler에 notify하도록 한다. Task Manager는 Error Handler를 참조하여 Error 발생시 현재 수행중인 Translate Task Thread를 종료하고 State 변경을 통해 즉각적으로 Error에서 회복하도록 한다.



그림 . Error Handler Component & Connector View

* 1. 시스템의 Risk & Management
     1. 사용성에 의한 성능 Risk

본 시스템은 ADL Language를 정의하고 이를 사용해서 즉각적으로 손쉽게 Architecture를 그릴 수 있도록 하는데 그 목적이 있다. 즉 ADL 작성의 편의성이 가장 높을 품질속성으로 이에 따른 성능적인 Tradeoff가 있다.

* 후보구조 1. Element - Relation으로만 구성된 ADL
* 후보구조 2. 기존 Markup language의 문법 사용

상기 두 후보구조는 공존할 수 없는 구조이며, Translate 성능적으로는 후보구조 2가 우수하나 후보구조 1이 사용성이 더 높아서 성능의 Risk를 감안하고 후보구조 1을 선택하였다. 그리고 이를 상쇄하기 위해 다양한 성능향상을 위한 구조를 적용하였지만, 근본적인 성능 Risk가 존재한다. 이를 해소하기 위해서는 본 시스템을 사용하는 고객들에게 A / B Test 등을 통해 사용이 편리하면서도 Parsing 성능이 좋은 ADL을 평가받고 지속 발전시켜야 할 것이다.

* + 1. 성능에 의한 변경용이성 Risk

본 시스템에서 Translate 성능은 사용성에 이어서 두번째로 높은 우선순위를 가지는 품질 속성이다. 이에 따라 성능과 변경 용이성 사이에 Tradeoff가 발생한다.

* 후보구조 21. Interface Layer를 분리 Publish – Subscriber Pattern 적용
* 후보구조 22. Task 기반의 Event Bus 적용

상기 두 구조는 양립할 수 없는 구조이며, 후보구조 22가 ADL Framework API 확장 시 보다 유연하게 적용이 가능한 구조이나 성능상의 문제로 후보구조 21을 선택하였다. 이로 인해 관련 Module에 수정이 필요한 상황이 발생하면 수정을 위한 M/M이 좀 더 발생할 수 있는 Risk가 존재한다. 하지만 후보구조 21도 상대적으로 변경 용이성이 떨어지는 구조이지, 변경 용이성이 낮은 구조는 아니다. 즉, 현 상황에서 선택할 수 있는 최선의 결정이며 수정이 필요한 상황이 발생하면 수정을 위한 M/M을 좀 더 할당하는 것이 해결책이 될 수 있다.

1. 최종 구조 평가(ATAM)

// 활동10. 최종 구조 평가

// 점검10-1. 구조에 영향을 미치는 품질에 대한 검토가 충분한가?

// 점검10-2. 설계 결정사항의 식별이 충분한가?

// 점검10-3. 설계 결정사항의 분석이 적절한가? (근거)

// 점검10-4. 최종 구조의 평가가 적절한가? (위험)