|  |
| --- |
| <<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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1. 시스템 개요 3](#_Toc68457810)

[1.1. 비즈니스 환경 3](#_Toc68457811)

[1.2. 시스템의 정의 5](#_Toc68457812)

[2. 요구사항 6](#_Toc68457813)

[2.1. 기능적 요구사항 6](#_Toc68457814)

[2.2. 비기능적 요구사항 8](#_Toc68457815)

[2.3. 품질 속성 9](#_Toc68457816)

[3. 시스템 구조 11](#_Toc68457817)

[4. 모듈 사양 12](#_Toc68457818)

[부록 13](#_Toc68457819)

[A. 도메인 모델 15](#_Toc68457820)

[B. 품질 시나리오 19](#_Toc68457821)

[C. 품질 시나리오 분석 21](#_Toc68457822)

[D. 후보 구조 23](#_Toc68457823)

[E. 후보 구조 평가 44](#_Toc68457824)

[F. 최종 구조 설계 50](#_Toc68457825)

[G. 최종 구조 평가(ATAM) 54](#_Toc68457826)

# 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.

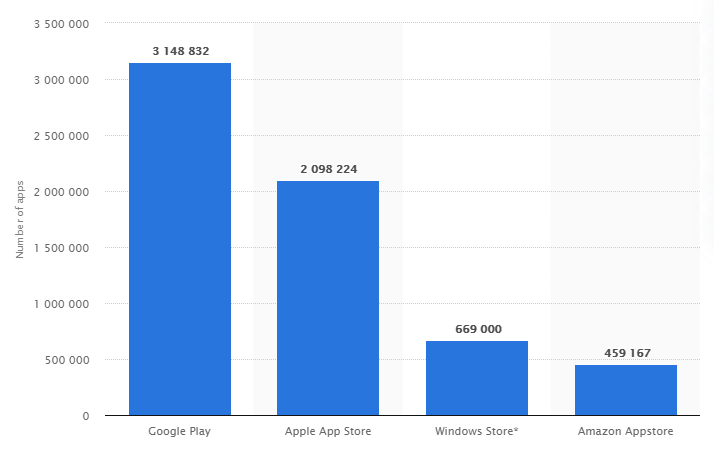


그림 . [Number of apps available in leading app stores as of 4th quarter 2020](https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores)

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

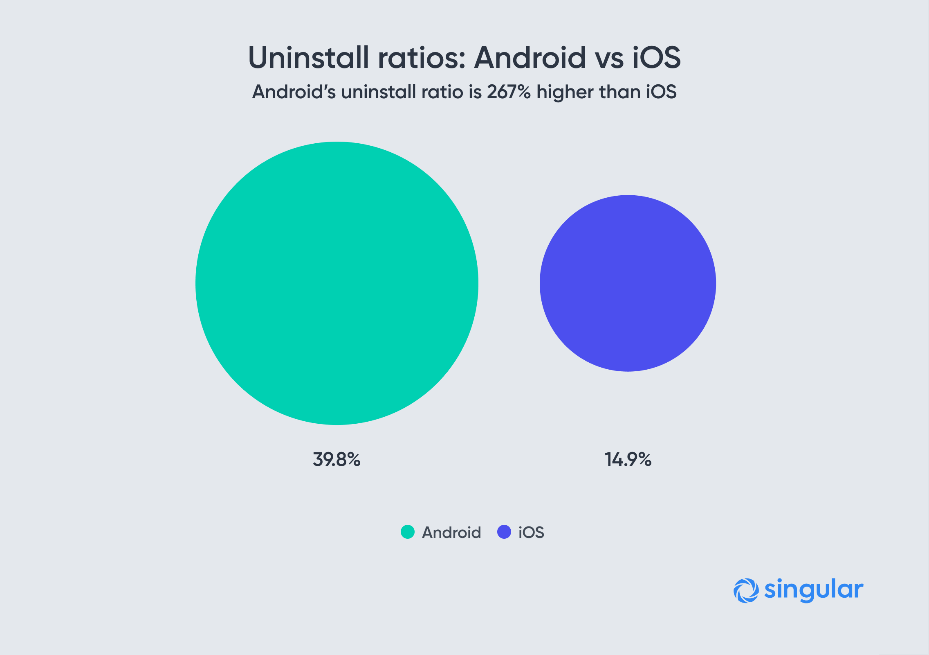


그림 . [Android uninstall rate is 2.5X more than iOS](https://www.singular.net/blog/app-uninstall-rates-and-coronavirus-lockdown-is-bad-for-retention/)

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.

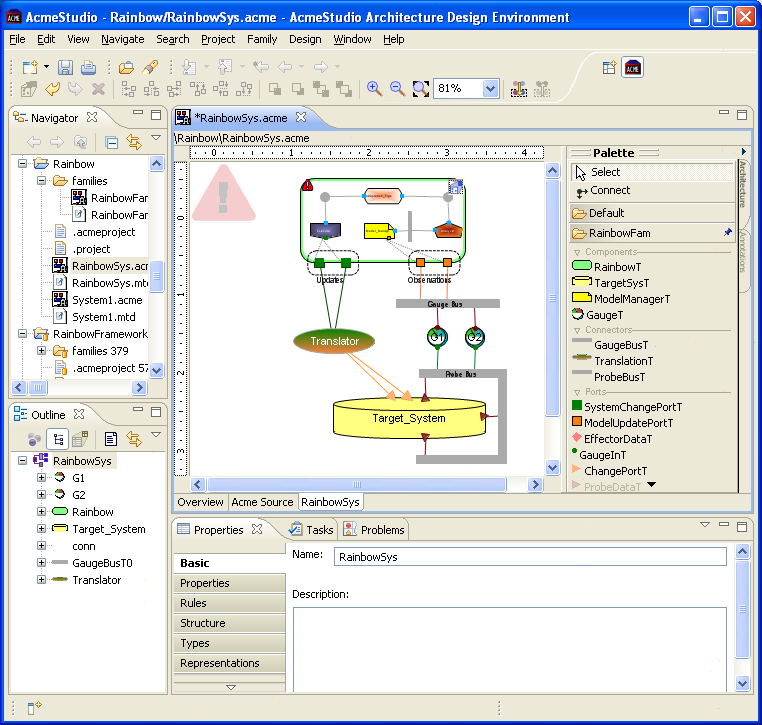


그림 . ADL 도구 예제(AcmeStudio)

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 ADL 도구 개발자를 위한 ADL Framework으로, ADL 도구 개발자가 본 시스템을 활용해서 ADL 도구를 개발하면 사용자가 ADL 도구를 사용해서 구조를 설계할 수 있도록 한다. 전체 구조는 아래와 같다.

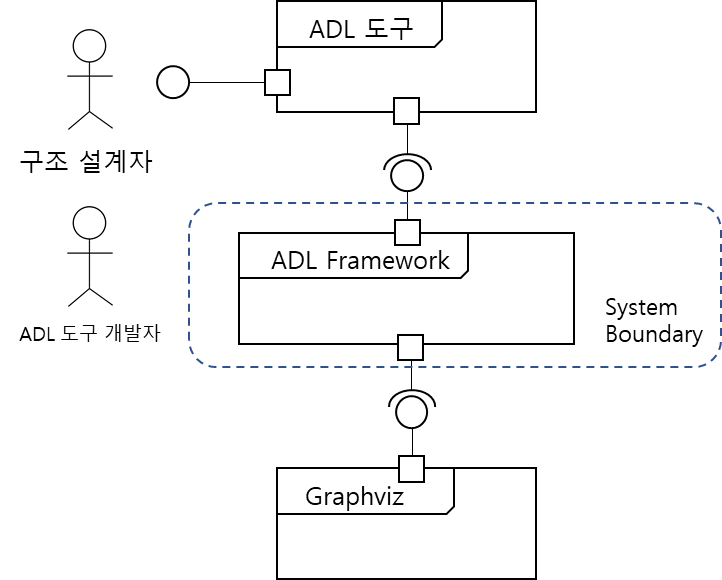


그림 . 시스템 정의

구조 설계자가 ADL 도구를 통해 본 시스템이 정의한 ADL을 작성하면 ADL Framework은 DOT([DOT (graph description language) - Wikipedia](https://en.wikipedia.org/wiki/DOT_(graph_description_language)))과 같은 graph description language로 변환을 하고, 실제 표현을 위해 Graphviz([Graphviz - Wikipedia](https://en.wikipedia.org/wiki/Graphviz))를 활용해서 Visualization한다.

# 요구사항

## 기능적 요구사항

ADL Framework의 Use Case는 다음과 같다.



그림 . Use Case Diagram

본 시스템은 ADL(Architecture Description Language)을 정의하며 Android Architecture 최적 설계를 위해 1. Component & Connector view와 2. Deployment View에 대한 Coding 방법을 명세한다.

1. Component & Connector view는 Android의 4대 Component인 Activity, Service, Broadcast Receiver, Content Provider를 Component로 추가하고 각 관계를 Connector로 기술하는 View이며
2. Deployment View는 외부 System 및 Library를 Component로 추가하고, Android Application의 대표적인 Architecture style인 MVVM, MVC, MVP의 각 파트를 Sub System으로 설정하고 해당 파트내에 Component들을 배치하는 형태의 View이다.

|  |  |
| --- | --- |
| **UC\_01** | **ADL Code를 작성한다.** |
| 설명 | Architecture Description Language를 정의한다. |
| 행위자 | 구조 설계자 |
| 선행조건 | 1. ADL은 Component & Connector View를 명세한다. 2. ADL은 Deployment View를 명세한다. 3. ADL은 Component를 명세한다. 4. ADL은 Component의 Stereo Type을 명세한다. 5. ADL은 Component의 Attribute(Label, Shape, Style, Color)를 명세한다. 6. ADL은 Component의 Group 설정을 명세한다. 7. ADL은 Connector를 명세한다. 8. ADL은 Connector의 from / to Component를 명세한다. 9. ADL은 Connector의 Type을 명세한다. 10. ADL은 Connector의 Attribute(Label, Color)를 명세한다. |
| 후행조건 |  |
| 기본 동작 | 1. 구조 설계자는 ADL Code를 작성한다. |
| 추가 동작 |  |

|  |  |
| --- | --- |
| **UC\_02** | **Translate** |
| 설명 | 작성된 ADL code를 Graph Image로 변경해서 반환 |
| 행위자 | ADL 도구 |
| 선행조건 |  |
| 후행조건 |  |
| 기본 동작 | 1. ADL 도구가 시스템으로 Translate를 요청한다. 2. 시스템은 ADL Code를 Parsing한다.    1. 시스템은 Component 및 Connector, Deployment를 Code에서 구분하다.    2. 시스템은 각 Component Type에 대응하는 DOT keyword로 변경한다.    3. 시스템은 각 Component의 Group에 대응하는 DOT keyword로 변경한다.    4. 시스템은 변경된 DOT keyword를 순서에 맞게 DOT Code를 생성한다. 3. 시스템은 작성한 DOT code를 Graphviz로 변환 요청한다. 4. Graphviz는 변환된 Graph image를 반환한다. 5. 시스템은 ADL도구로 Graph image를 반환한다. |
| 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. System은 Component & Connector View를 확인한다.    1. System은 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.    2. System은 Connector의 ‘From’에 해당하는 Component에 대응하는 Class에 ‘To’에 해당하는 Component에 대응하는 Class를 Class member 변수로 추가한다. 3. System은 Deployment View를 확인한다.    1. System은 Group의 Type을 확인한다.       1. System은 Group의 Type(model, view, viewmodel, control, presenter)에 따라 package를 생성한다.    2. 2.의 방식으로 각 package 내부에 Component를 추가한다. |
| 추가 동작 |  |

## 비기능적 요구사항

|  |  |  |
| --- | --- | --- |
| **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. | |

## 품질 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 DOT Code 문자 수]  작을수록 좋다. | |

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

|  |  |  |
| --- | --- | --- |
| **QA\_03** | **Performance** | **Translate 요청 시 첫 Graph 반환 시간** |
| 설명 | 시스템은 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** | **Component Type 변경의 용이성** |
| 설명 | Component Type 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 환경 | Component Type 변경 요청 상황 | |
| 자극 | 고객이 개발팀에게 Component Type 변경을 요청한다. | |
| 반응 | 개발팀은 추가된 Component 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** | **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

// 활동8. 구조 명세

// 점검8-1. 프로세스 등의 Allocation이 적절한가?

// 점검8-2. 컴포넌트 측면에서 Grouping이 적절한가?

// 점검8-3. 시스템의 동작 특성에 대한 설명이 적절한가?

# 모듈 사양

// 활동9. 모듈 명세

// 점검9-1. 컴포넌트 명세가 충분한가? (개발 가능)

// 점검9-2. 모듈 측면에서 Grouping이 적절한가?

// 점검9-3. Work Assignment가 적절한가?

// 점검9-4. 개발 측면에서 구조적 특징에 대한 설명이 적절한가?

부록

[A. 도메인 모델 16](#_Toc68457827)

[B. 품질 시나리오 20](#_Toc68457828)

[C. 품질 시나리오 분석 22](#_Toc68457829)

[D. 후보 구조 24](#_Toc68457830)

[D1. QA\_01 ADL Code 작성의 편의성 24](#_Toc68457831)

[D1.1. 후보구조 1. Element - Relation으로만 구성된 ADL 24](#_Toc68457832)

[D1.2. 후보구조 2. ADL을 따로 정의하지 않고 기존 ADL을 사용 29](#_Toc68457833)

[D1.3. 후보구조 3. 기존 Markup language의 문법을 그대로 사용 29](#_Toc68457834)

[D2. QA\_02 ADL to DOT 변환 시간 29](#_Toc68457835)

[D2.1. 후보구조 4. ADL을 DOT Language와 동일하게 정의 29](#_Toc68457836)

[D2.2. 후보구조 5. Dot language Parser library를 사용 29](#_Toc68457837)

[D2.3. 후보구조 6. ADL Code분석을 단계별로 나누어 동작 30](#_Toc68457838)

[D2.4. 후보구조 7. ADL Code 분석을 Pipe & filter 방식으로 수행 34](#_Toc68457839)

[D2.5. 후보구조 8. Update를 위해 Token 및 Parse Tree 유지 35](#_Toc68457840)

[D3. QA\_03 Translate 요청 시 첫 Graph 반환 시간 35](#_Toc68457841)

[D3.1. 후보구조 9. 연속적으로 Graph를 갱신 36](#_Toc68457842)

[D3.2. 후보구조 10. 일정한 주기로 Graph를 갱신 36](#_Toc68457843)

[D3.3. 후보구조 11. Translate 요청전에 Background parsing 37](#_Toc68457844)

[D4. QA\_04 Component Type 변경 용이성 37](#_Toc68457845)

[D4.1. 후보구조 12. 별도의 Component Type library로 발행 38](#_Toc68457846)

[D4.2. 후보구조 13. Decorate Pattern을 적용 38](#_Toc68457847)

[D4.3. 후보구조 14. Strategy Pattern을 적용 39](#_Toc68457848)

[D5. QA\_05 Code 생성 Language 확장성 40](#_Toc68457849)

[D5.1. 후보구조 15. Code Generation Library 적용 40](#_Toc68457850)

[D5.2. 후보구조 16. Language별 Class Prototype변경 42](#_Toc68457851)

[D6. NFR\_01. ADL Parsing 중 Error 발생시 회복 시간 43](#_Toc68457852)

[D6.1. 후보구조 17. ADL 분석 단계에서 Error 발생시 Thread 종료 44](#_Toc68457853)

[E. 후보 구조 평가 45](#_Toc68457854)

[E1. QA\_01 ADL Code 작성의 편의성에 관한 후보구조 평가 45](#_Toc68457855)

[E2. QA\_02 ADL Code의 DOT Code 변환성능에 관련한 후보구조 평가 46](#_Toc68457856)

[E3. QA\_03 Translate 요청 시 첫 Graph 반환 성능 관련 후보구조 평가 47](#_Toc68457857)

[E4. QA\_04 Component Type 변경 용이성 후보구조 검토 48](#_Toc68457858)

[E5. QA\_05 Code 생성 Language 확장성 후보구조 검토 48](#_Toc68457859)

[E6. NFR\_01 ADL Parsing중 Error 발생시 회복 시간 후보 구조 검토 49](#_Toc68457860)

[E7. 후보 구조 평가 결과 50](#_Toc68457861)

[F. 최종 구조 설계 51](#_Toc68457862)

[F1. 시스템의 변경용이성 측면 52](#_Toc68457863)

[F2. 시스템의 Translate 성능 측면 52](#_Toc68457864)

[F3. 시스템의 신뢰성 측면 54](#_Toc68457865)

[F4. 시스템의 Risk & Management 54](#_Toc68457866)

[G. 최종 구조 평가(ATAM) 55](#_Toc68457867)

1. 도메인 모델

본 과제의 도메인 모델은 아래 그림과 같다.

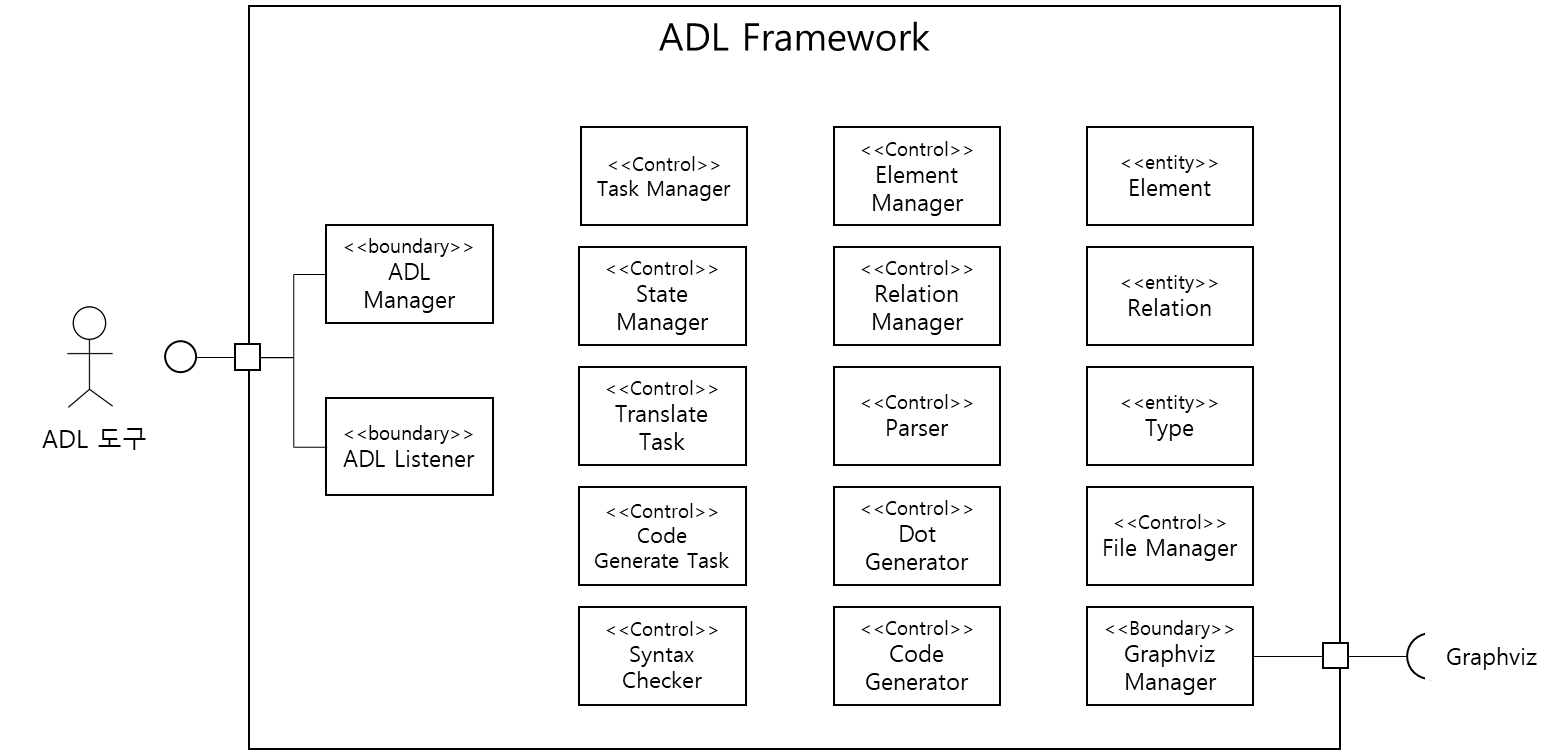


그림 . Domain Model

각 Use Case에 대한 세부적인 Sequence Diagram은 아래와 같다.



그림 . UC\_02 Translate (1/2) - Parsing



그림 . UC\_02 Translate (2/2)



그림 . UC\_03 Code Generate

1. 품질 시나리오

본 시스템의 품질 시나리오 들은 아래와 같다.

|  |  |  |
| --- | --- | --- |
| **QS\_01** | **Performance** | **ADL Code의 DOT Code 변환 시간** |
| 설명 | 시스템은 ADL Code의 DOT language로 변환이 빨라야 한다. | |
| 측정 | [변환시간] = [DOT Code 완성 시간] – [Translate 요청 시간] | |
| **QS\_02** | **Performance** | **Translate 요청 시 첫 Graph 반환 시간** |
| 설명 | 시스템은 ADL Code의 LOC에 관계없이 Translate 요청 시 Graph 반환이 빨라야 한다. | |
| 측정 | [반환시간] = [처음으로 Graph를 반환한 시간] – [Translate 요청시간] | |
| **QS\_03** | **Performance** | **Code Generate시 Code 생성 시간** |
| 설명 | Code Generate시 Code 생성이 빨라야 한다. | |
| 측정 | [코드 생성시간] = [생성 완료시간] – [Code Generate 요청시간] | |
| **QS\_04** | **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\_05** | **Modifiability** | **Component Type 변경 용이성** |
| 설명 | Component Type 변경 시 기존 Logic의 변경이 적어야 한다. | |
| 측정 | [변경률] = 영향을 받는(빌드 및 검증을 다시 해야 하는) 모듈 LOC / 전체 System LOC | |
| **QS\_06** | **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\_07** | **Reliability** | **ADL Parsing 중 Error 발생시 회복 시간** |
| 설명 | 시스템이 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\_08** | **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\_09** | **Usability** | **ADL Code 작성의 편의성** |
| 설명 | ADL Code작성이 쉽고 직관적이며 간결하게 작성되야 한다. | |
| 측정 | [간결성] = [ADL Code 문자 수] / [변환된 DOT Code 문자 수] | |

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 |
| Performance | QS\_01 | ADL Code의 DOT Code 변환 시간 | H | H | QA\_02 |
| QS\_02 | Translate 요청 시 첫 Graph 반환 시간 | H | H | QA\_03 |
| QS\_03 | Code Generate시 Code 생성 시간 | L | H |  |
| Compatibility | QS\_04 | 기존 ADL Tool과의 호환성 | L | H |  |
| Modifiability | QS\_05 | Component Type 변경 용이성 | H | M | QA\_04 |
| Interoperablility | OS\_06 | ADL 공유 작업 동시성 | L | H |  |
| Reliability | QS\_07 | ADL Parsing 중 Error 발생시 회복 시간 | H | L | NFR\_01 |
| Extendibility | QS\_08 | Code Generate Language 확장성 | M | H | QA\_05 |
| Usability | QS\_09 | ADL Code 작성의 편의성 | H | H | QA\_01 |

본 시스템은 ADL Language를 정의하고 이를 사용해서 즉각적으로 손쉽게 Android Application Architecture를 그릴 수 있도록 하는데 그 목적이 있다. 이에 따라 **QS\_09. 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 매우 높은 중요도를 가지며, Graph 변환의 중간단계인 **QS\_01. ADL Code의 DOT Code 변환 시간**을 **QA\_02**로 선정한다. 더불어 Translate 요청 시 Grape 반환시간은 전체 코드의 DOT 변환 이전에 Translate 요청에 즉각적으로 Graph를 반환하여 즉시 응답성을 확보할 필요가 있다. 이에 따라 **QS\_02 Translate 요청 시 첫 Graph 반환 시간**을 **QA\_03**로 선정한다.

**QS\_04 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.

Component Type은 요청에 따라 빈번히 변경이 이루어질 부분이어서 **QS\_05. Component Type 변경 용이성**은 본 시스템의 중요한 품질 속성이며 **QA\_04**로 선정한다.

하지만 **QS\_06 ADL 공유 작업 동시성**은 Architecture 설계과정이 일반적으로 공동 작업으로 진행되지 않는 점과 동시성 확보를 위한 비용이 큰 점을 이유로 품질속성에서 제외한다.

**QS\_07. 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.

**QS\_08. Code 생성 Language 확장성**은 Android Application이 Java가 아닌 Kotlin등의 프로그래밍언어로 확대되고 있는 상황인만큼 **QA\_05** 선정해서 추후 확장성을 확보한다.

**QS\_03 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.

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.

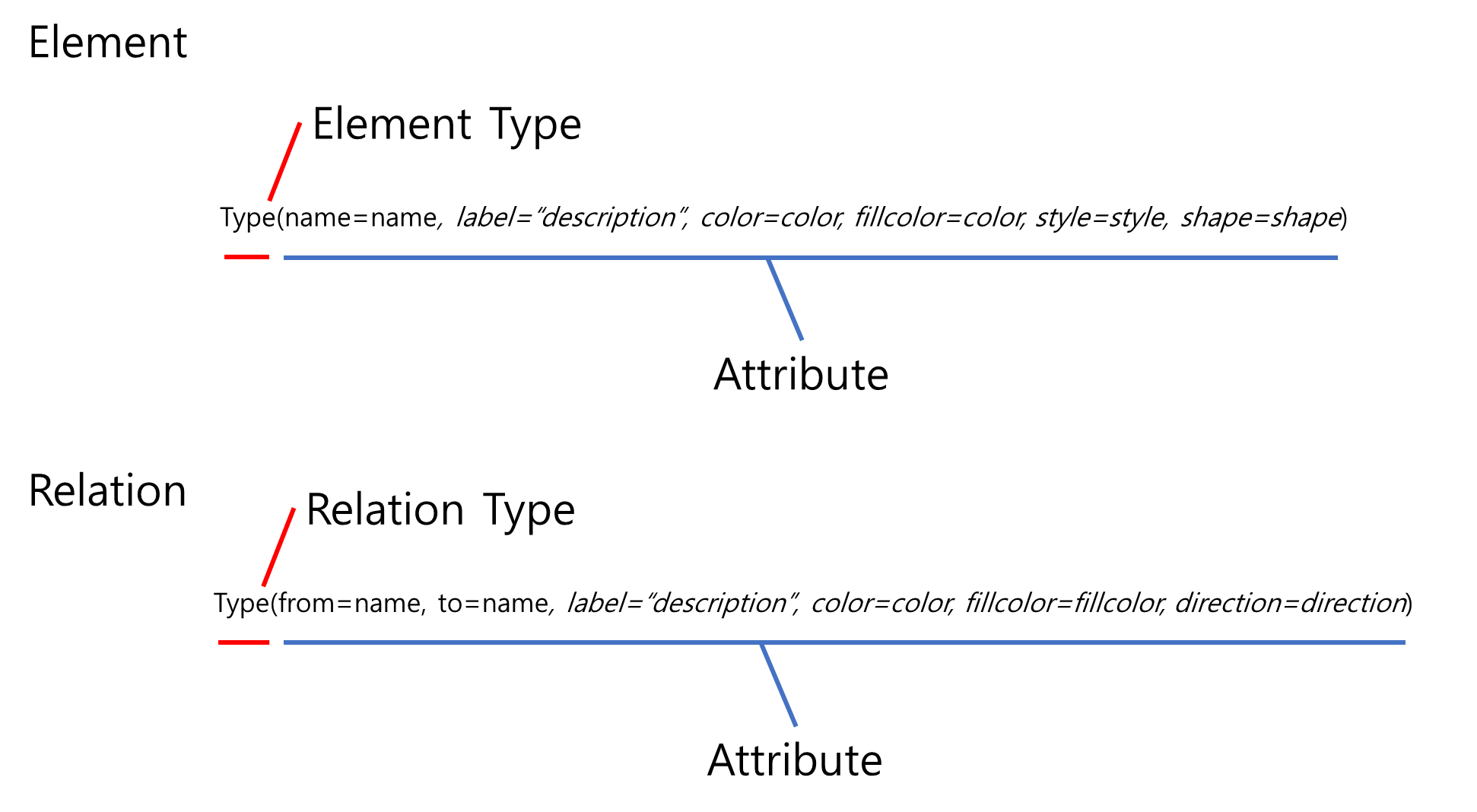


그림 . Element와 Relation 기술 방법

Element Type은 어떤 특성의 Component인지를 기술하며, Relation Type은 UML상의 Relation을 동일하게 표현한다.

|  |  |
| --- | --- |
| **Element** | **Relation** |
| ComponentAndConnector  Deployment  Activity  Service  BroadcastReceiver  ContentProvider  Model  View  Controller  Presenter  ViewModel  Component  Database  Cloud  Server  Process  Thread | Association  Dependency  Inheritance  Realization  Aggregation  Composition |

표 . Element와 Relation의 Type

Element와 Relation은 각각 아래와 같은 Attribute를 가질 수 있다.

|  |  |  |  |
| --- | --- | --- | --- |
| **Color, FillColor** | **Style** | **Shape** | **Direction** |
| “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 | solid  dashed  dotted  bold  invis  filled  diagonals  rounded | box  circle  ellipse  point | up  down  left  right |

표 . Element와 Relation의 Attributes

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

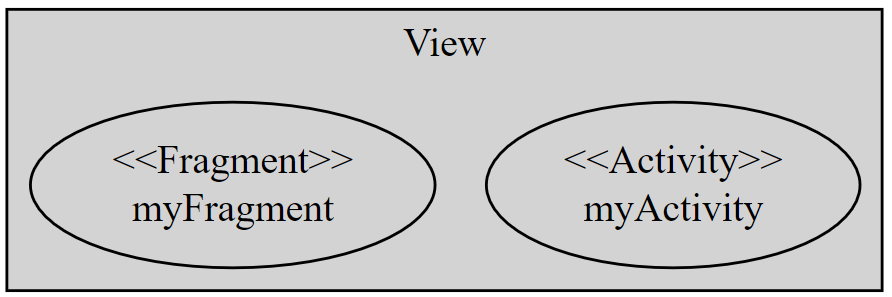


그림 . View의 Child로 선언된 Fragment와 Activity의 Graph

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.

|  |
| --- |
| Deployment(name=deploymentView, description=”Deployment View”)  View(name=view, fillcolor=RED)  Fragment(name=myFragment)  Activity(name=myActivity)  Association(myFragment, 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(myActivity, liveData)  Association(liveData, myService)  Aggregation(myCloud, myService) |

Code . MVVM Deployment Diagram ADL Code

본 시스템은 Graphviz를 사용해서 Graph를 생성한다. 이에 따라 Graphiviz에서 사용하는 DOT Language로 변환이 필요하며 위에서 작성한 MVVM Deployment Diagram은 아래와 같은 DOT code로 변경된다.

|  |
| --- |
| digraph G {  label = "Deployment 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 Deployment Diagram DOT code

변경된 DOT Code를 Graphviz에서 Graph로 변환한 결과는 아래와 같다.

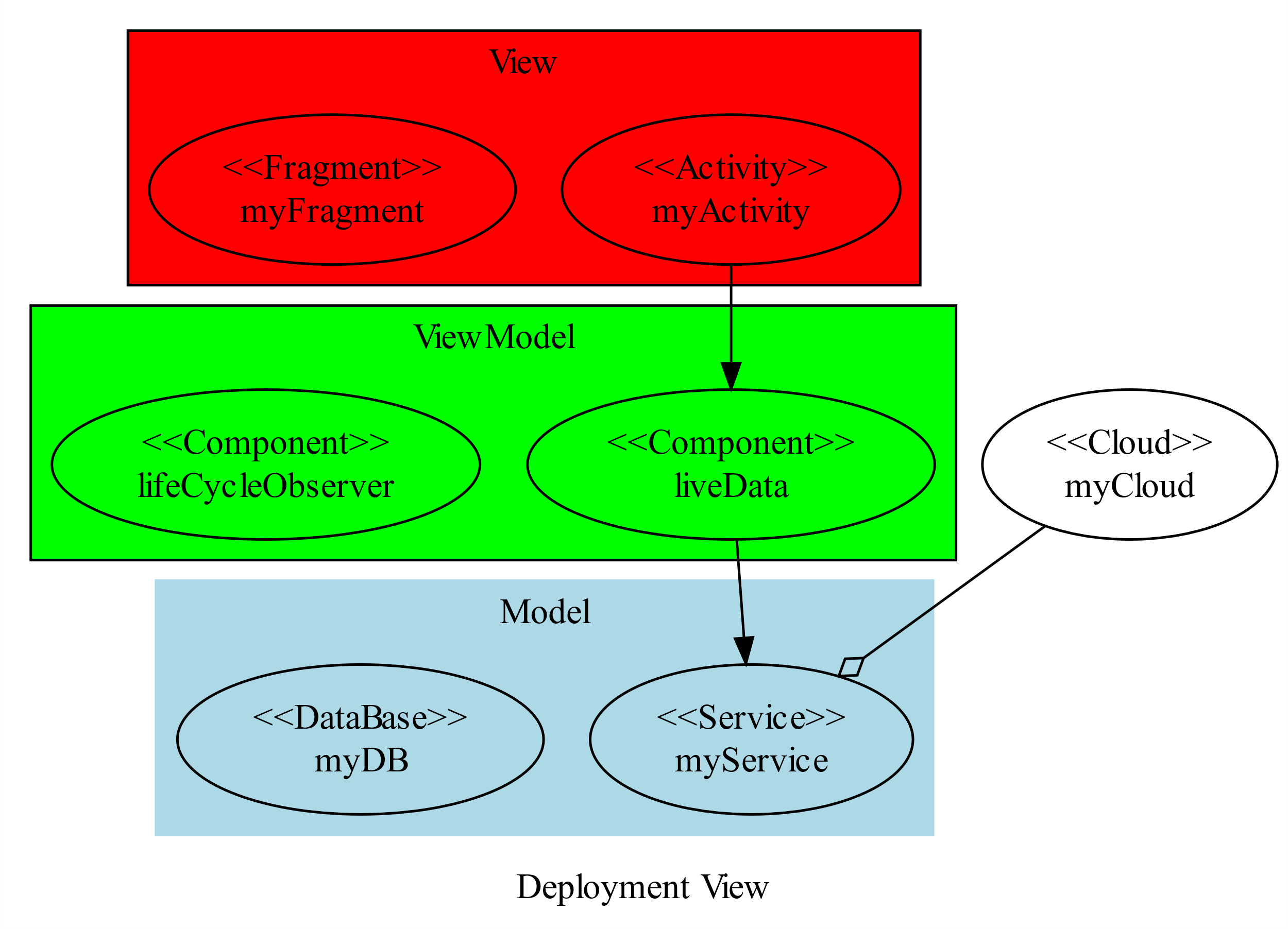


그림 . MVVM Deployment Diagram

* + 1. 후보구조 . ADL을 따로 정의하지 않고 기존 ADL을 사용

AcmeStudio같은 Tool에서 기 정의된 ADL이 존재한다. 이를 그대로 활용해서 별도로 ADL을 정의하는 비용을 절감하고, 기존 Tool에 익숙한 사용자는 새로운 language를 학습할 필요없이 그대로 사용할 수 있다.

* + 1. 후보구조 . 기존 Markup language의 문법을 그대로 사용

HTML이나 XML, Json과 같은 널리 사용되는 Markup Language의 문법을 그대로 적용해서 별도로 ADL을 정의하는 비용을 절감하고, 사용자로 하여금 ADL 학습에 대한 부담을 줄이고 익숙한 언어 그대로 사용할 수 있는 이점이 있다.

* 1. QA\_02 ADL to DOT 변환 시간

사용자는 ADL Code 작성후에 Graph로 변환시간이 빠를수록 만족도가 높을 것이다. 즉각적인 Graph 반환을 위해서는 Graphviz로 변환 요청하기 전 단계인 DOT 언어 생성이 빨라야 하며 ADL을 DOT로 빠른 시간안에 변환하는 후보 구조를 각각 검토한다.

* + 1. 후보구조 . ADL을 DOT Language와 동일하게 정의

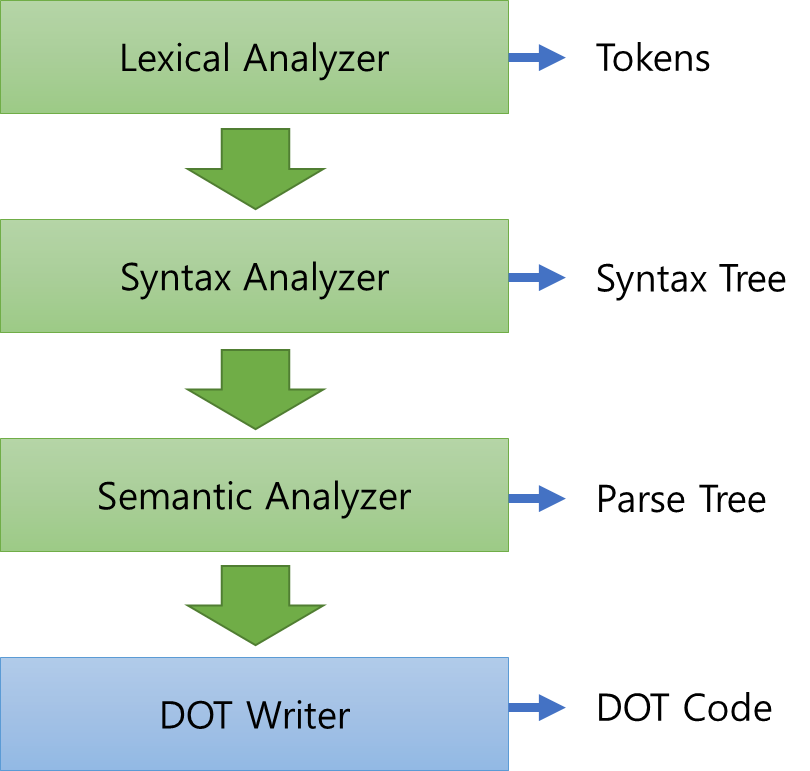
ADL을 DOT Language와 동일하게 정의하는 것을 후보 구조로 검토한다. ADL을 DOT로 변환하는 동작 자체를 제외할 수 있어서 변환시간 자체가 0이 될 수 있다.

* + 1. 후보구조 . Dot language Parser library를 사용

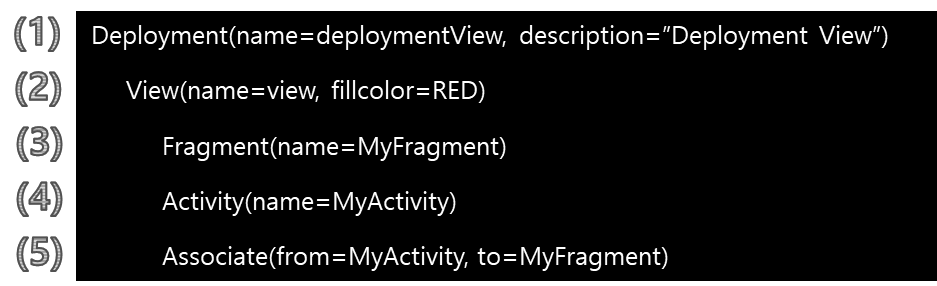
Python과 같은 기존 Programming language로 Code를 작성하면 Dot language로 변환하는 library들이 존재한다.([gvgen](https://github.com/stricaud/gvgen)) 해당 Library를 사용할 경우 Programming language를 그대로 사용할 수 있어서 Library의 사용 방법을 익히는 것 외에는 익숙한 Programming language를 그대로 사용할 수 있는 이점이 있다.

* + 1. 후보구조 . ADL Code분석을 단계별로 나누어 동작

ADL Code를 DOT Language로 변환하기 위해 어휘분석(lexical analyzer) 단계와 구문 분석(syntax analyzing) 단계, 그리고 의미 분석(semantic analyzer)으로 나누어서 동작한다. 별도의 단계 없이 변환하는 경우 변환중인 문자열의 의미를 분석하기 위해서는 전체 code상에서의 의미를 확인하기 위한 search동작이 빈번히 이루어지게 되는데, 단계를 두어 변환이 이루어지는 만큼 Keyword Searching의 시간을 절약할 수 있다. Batch Sequential Style로 각 단계를 진행하며 어휘 분석 단계에서는 Code의 각 문자를 확인해서 Token으로 변환하고, 구문 분석 단계 에서는 Token을 구조를 가진 구문 트리(syntax tree)로 변환한다. 그리고 의미 분석단계 에서는 프로그램의 의미(semantic)에 따라 필요한 정보를 유추/분석해서 Parse Tree를 생성하고, 이를 통해 DOT Language로 변환한다.

  
그림 . 구분된 분석 단계

예를 들어 하기 후보구조1에서 정의한 ADL을 변환한다고 할 때



Code . ADL Code

어휘 분석 단계에서는 Tokenizer를 통해 각 문자열을 Token으로 분류하고,

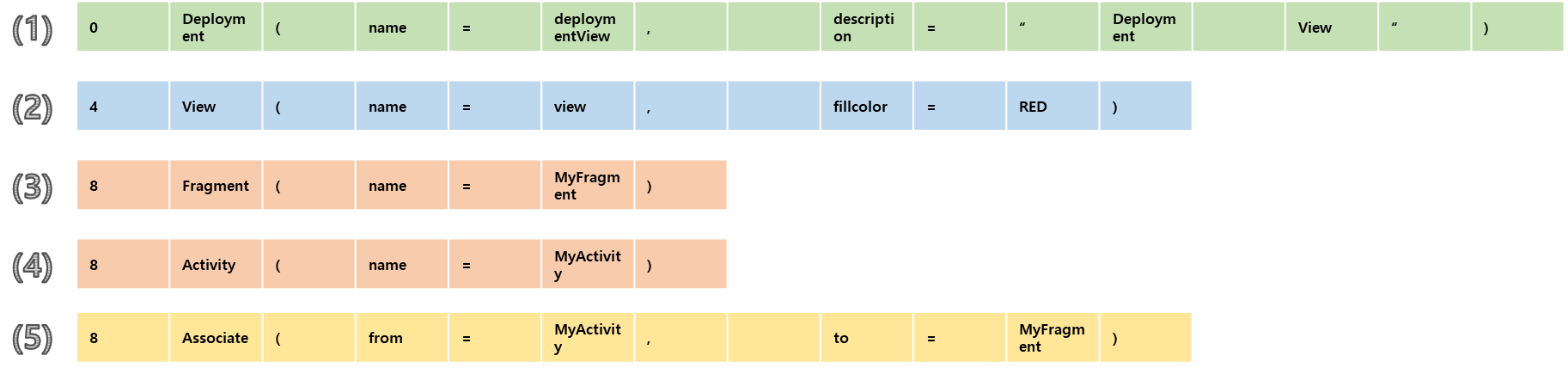


그림 . Token

이후 구문 분석 단계에서는 Syntax Tree를 각 문자열별로 생성하며,

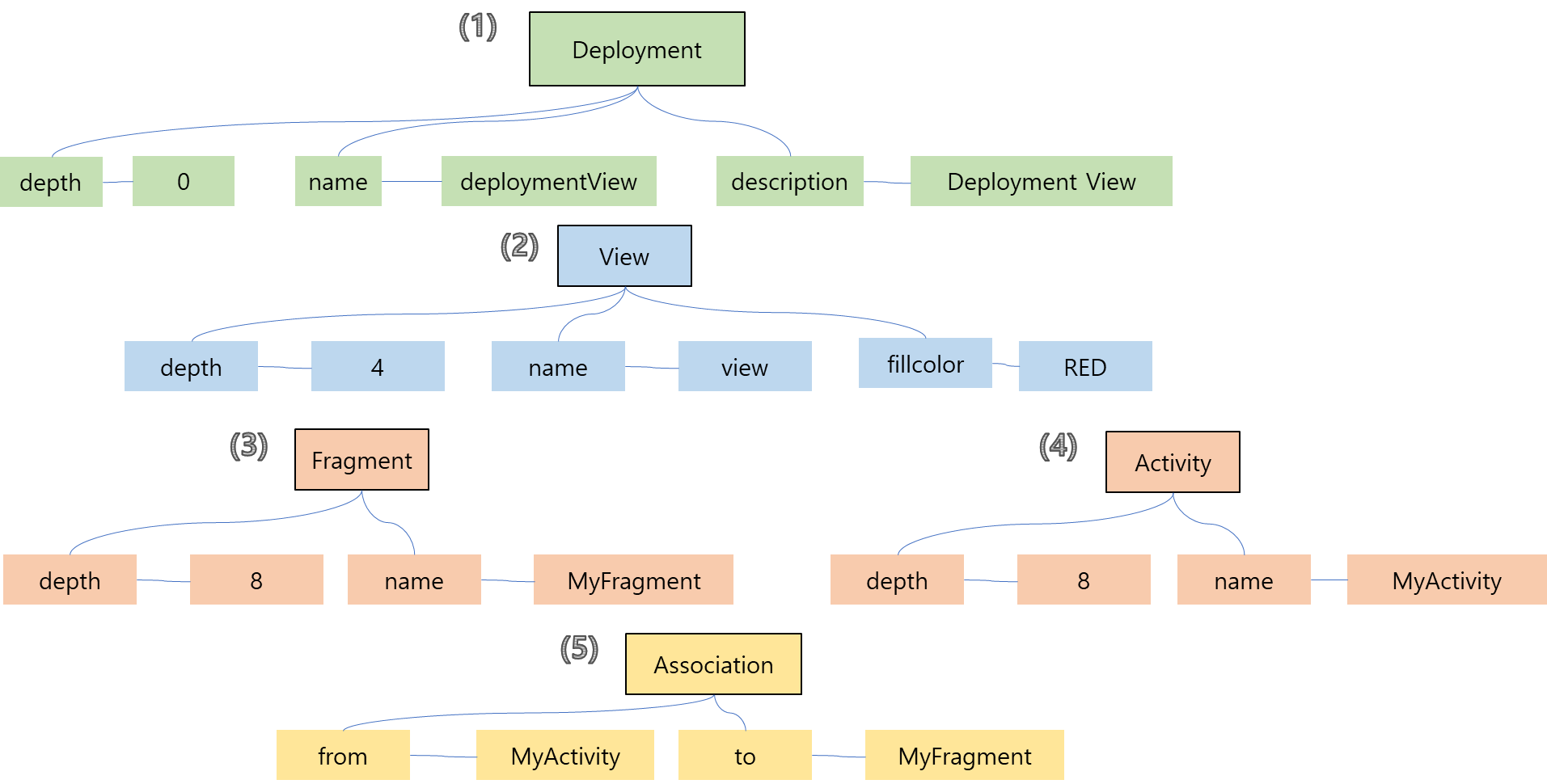


그림 . Syntax Tree

이후 의미 분석 단계 에서는 각 Syntax들의 연관 관계를 추가해서 Parse Tree를 만들고,

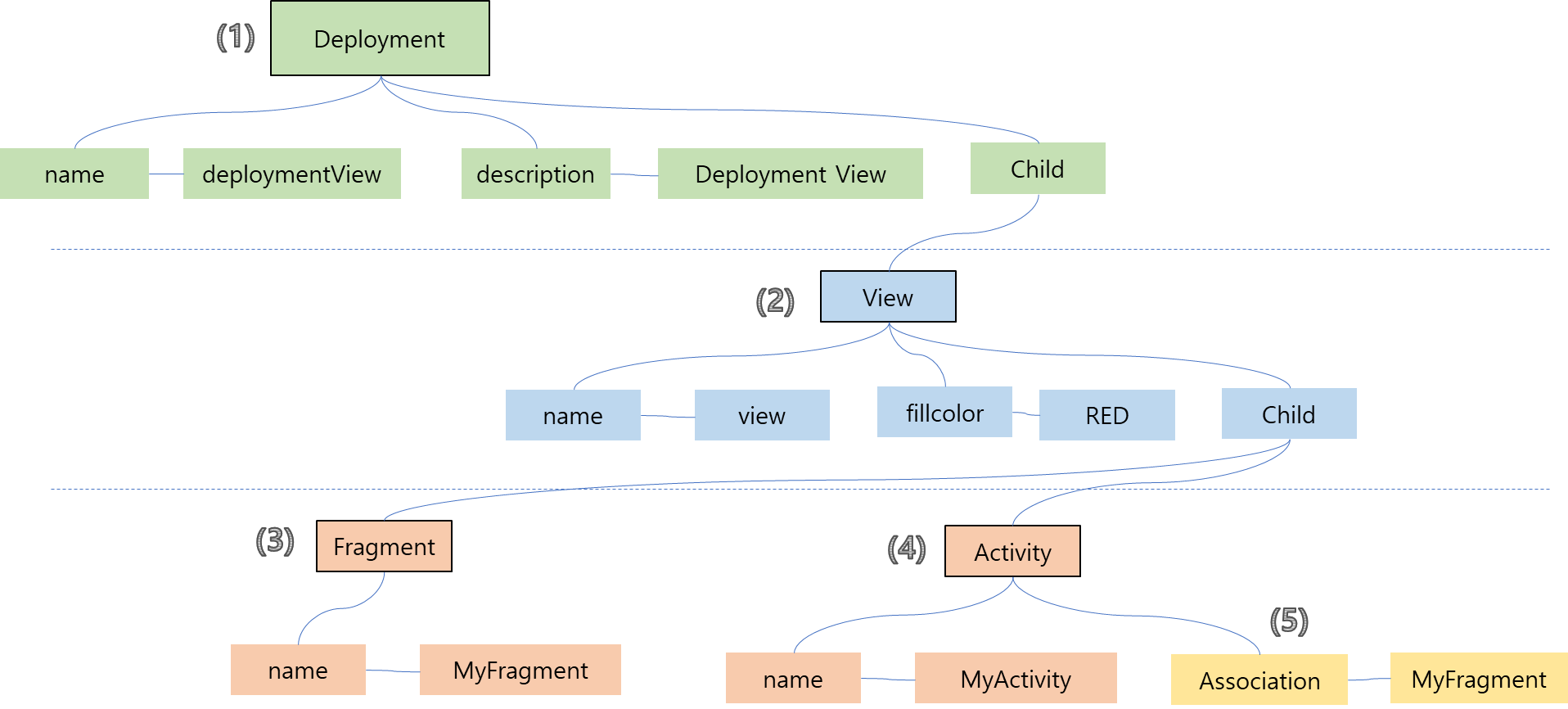


그림 . Parse Tree

Parse Tree가 완성되면 MappingTable을 참조해서 DOT code를 생성한다.

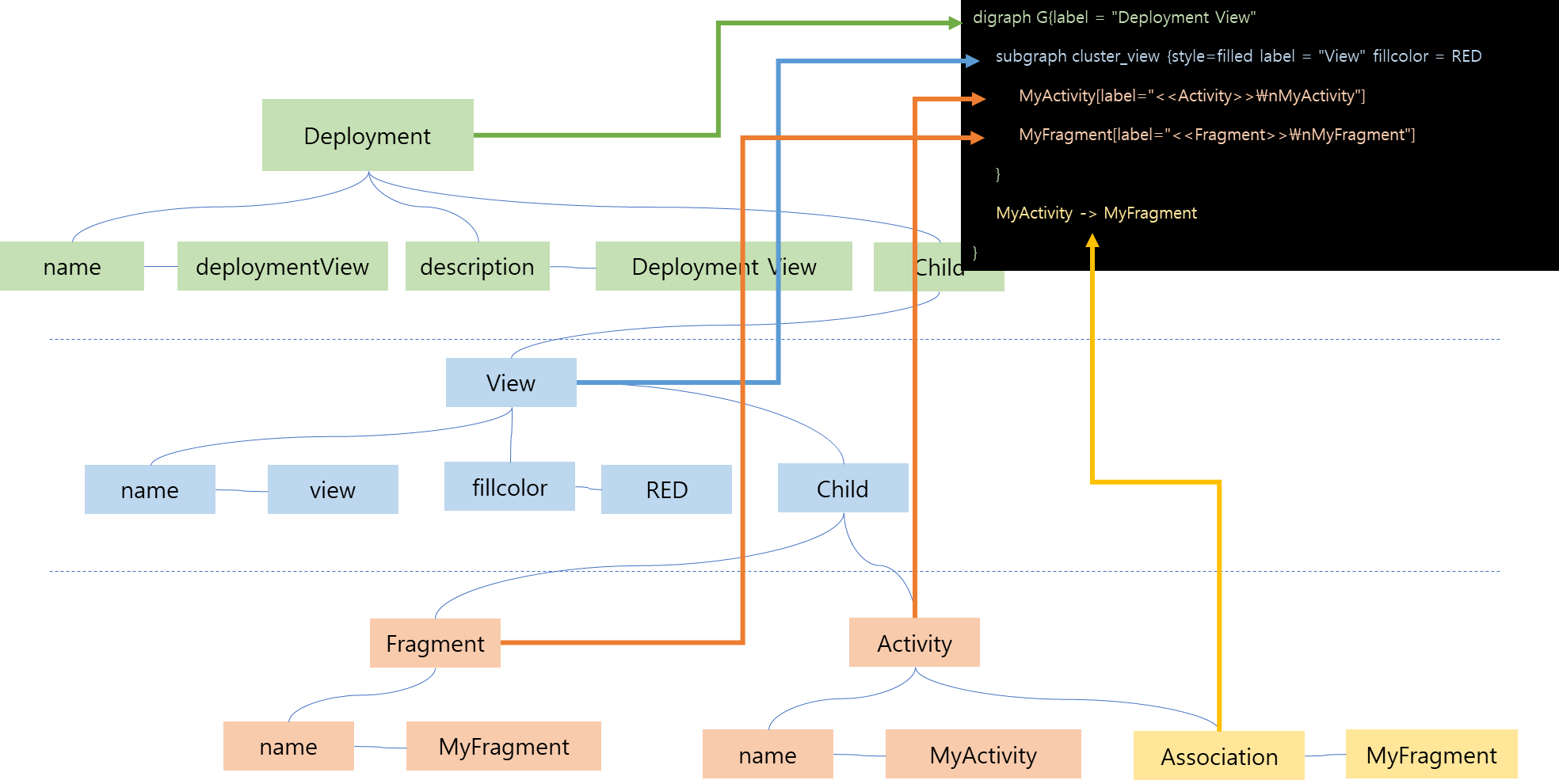
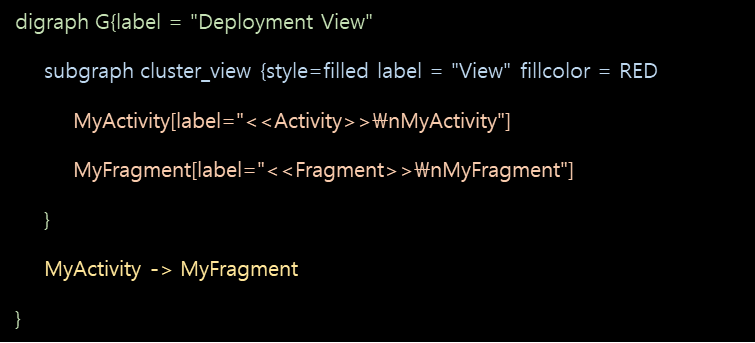


그림 . DOT Code

예시 ADL Code에 대한 최종 DOT Code는 아래와 같다.



Code . DOT Code

생성한 DOT Code를 Graphviz를 통해 Graph 변환을 요청하면 하기와 같은 Graph를 얻게 된다.

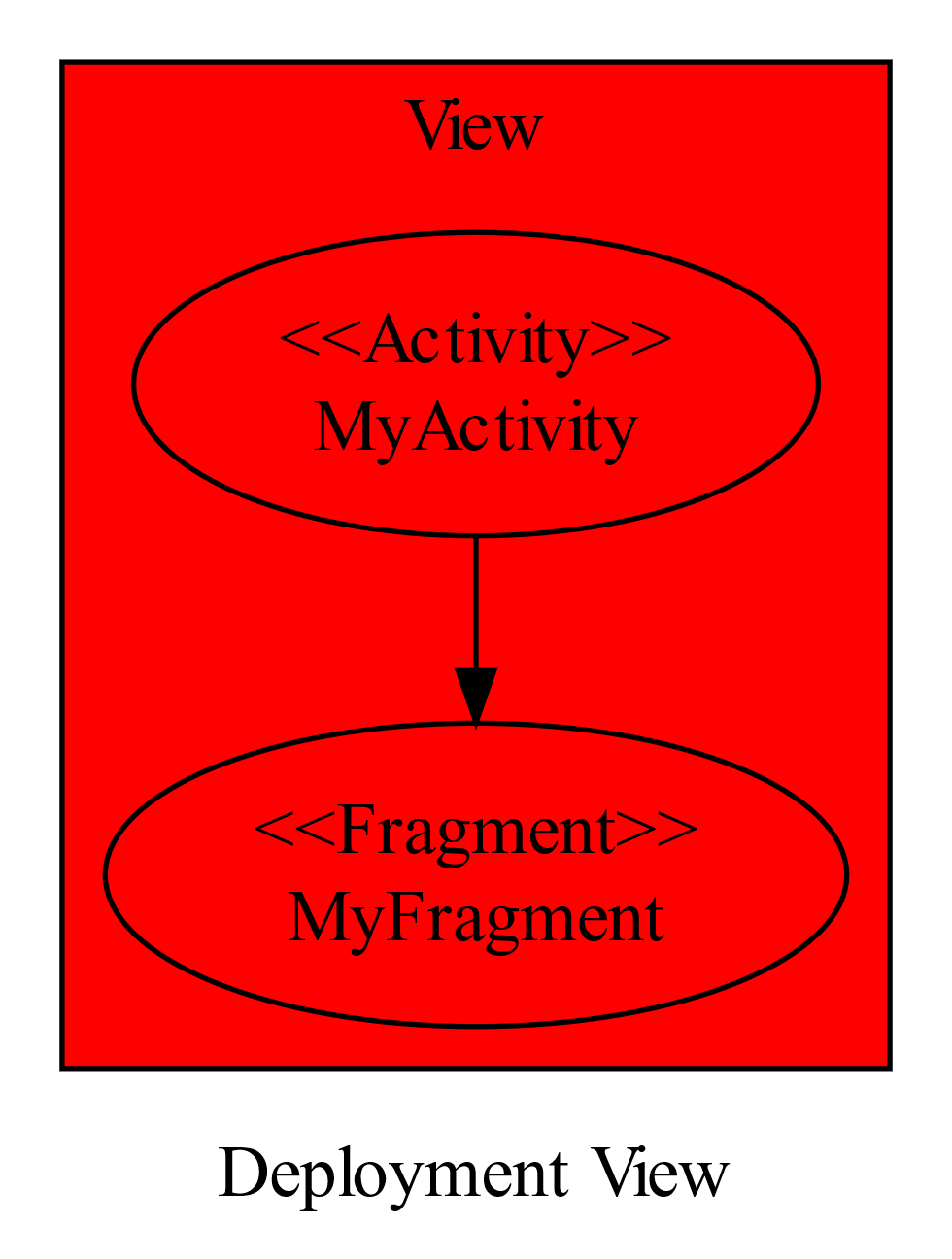


그림 . Graphviz Graph

* + 1. 후보구조 . ADL Code 분석을 Pipe & filter 방식으로 수행

Pipe & Filter구조를 적용해서 후보구조6에서 구분한 변환 동작을 각각의 Thread로 병렬 처리하게 되면 DOT Code 변환 시간을 줄일 수 있다.

1. 어휘분석 단계에서 한 Line의 Tokenizing이 끝나게 되면 해당 Token을 바로 구문 분석 Queue로 push하고
2. 구문분석에서 해당 Token에 대한 Syntax Tree를 생성하면 바로 의미 분석 Queue로 push하고,
3. 의미분석에서 전달된 Syntax Tree를 기존 Parse Tree에서의 depth를 고려해서 전체 계층을 구성한다. (전체 계층에 대한 구조를 가지고 있는 것은 Parse Tree뿐이다. Token과 Syntax Tree를 Parse Tree에 추가되면 바로 소멸된다.)

Parse Tree가 완성되면 이를 기준으로 DOT Mapping Table을 참조해서 DOT Document를 생성한다. 이 각각의 filter들을 별도 Thread로 구성한다. 본 후보구조는 ADL문법이 변경되는 경우에도 해당하는 filter만 변경하면 돼서 변경용이성에도 이점이 있다.

* + 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. 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 DOT로 변환하는 과정, 3. DOT를 Graph로 변환하는 과정이 있다. 이중 3에 해당하는 과정은 본 시스템의 바운더리 밖의 동작으로 Graphviz 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 DOT Code를 Graphviz에 변환 요청하여 Graph를 계속해서 update해준다. 이를 위해서는 2번 과정(분석된 ADL을 DOT로 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 9은 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 DOT Code 생성 및 Graph 변환을 진행한다. 이후 update되는 Parse Tree는 DOT Writer로 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 DOT 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 Component Type 변경 용이성

본 시스템이 정의한 Component Type은 Android Application에 맞게 ‘Activity’, ‘Service’, ‘Content Provider’, ‘Broadcast Receiver’ 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. Component Type의 추가 / 삭제 / 변경 시 시스템의 큰 변경 없이 적용이 가능해야 하며 이에 용이한 후보구조를 검토한다.

* + 1. 후보구조 . 별도의 Component Type library로 발행

Component Type의 Concrete Module을 ADL Framework에서 분리한다. Component Type의 추가 / 삭제 / 변경이 필요한 경우에 ADL Framework을 다시 배포할 필요없이 Component Type library만 배포함으로써 적용이 가능하다. Component Type을 참조하기 위한 Interface는 ADL Framework이 그대로 가지고 있기 때문에 ADL Framework의 변경사항은 없으며, 변경이 필요한 Component Type library의 수정만으로 적용이 가능하다.



그림 . Component Type의 Library 분리

* + 1. 후보구조 . Decorate Pattern을 적용

Component Type에 Decorate Pattern이 적용되어 있으면, Type을 추가 / 삭제 / 변경 시 기존 Module의 변경 없이 Decorate Module 추가만으로 적용이 가능하다.



그림 . Component Type with Decorated pattern

* + 1. 후보구조 . Strategy Pattern을 적용

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



그림 . Component Type with Strategy pattern

* 1. QA\_05 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.

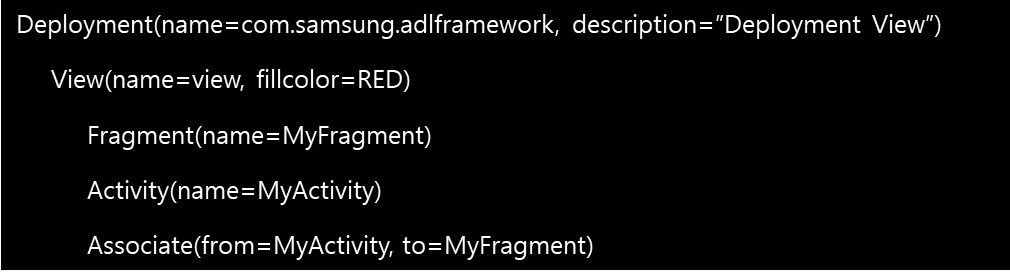
* + 1. 후보구조 . Code Generation Library 적용

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 Component Type별 Mapping Class Table이 필요한데, 이에 따라 후보구조 12에 의존성이 있으며 Component Type과 함께 배포되어야 한다.



그림 . Code Generation Library

예를 들어 아래와 같은 ADL Code가 있다면 Component Type에 해당하는 Class를 인지하기 위해 Component Type Table이 필요하다.



Code . Code Generate를 위한 ADL Code

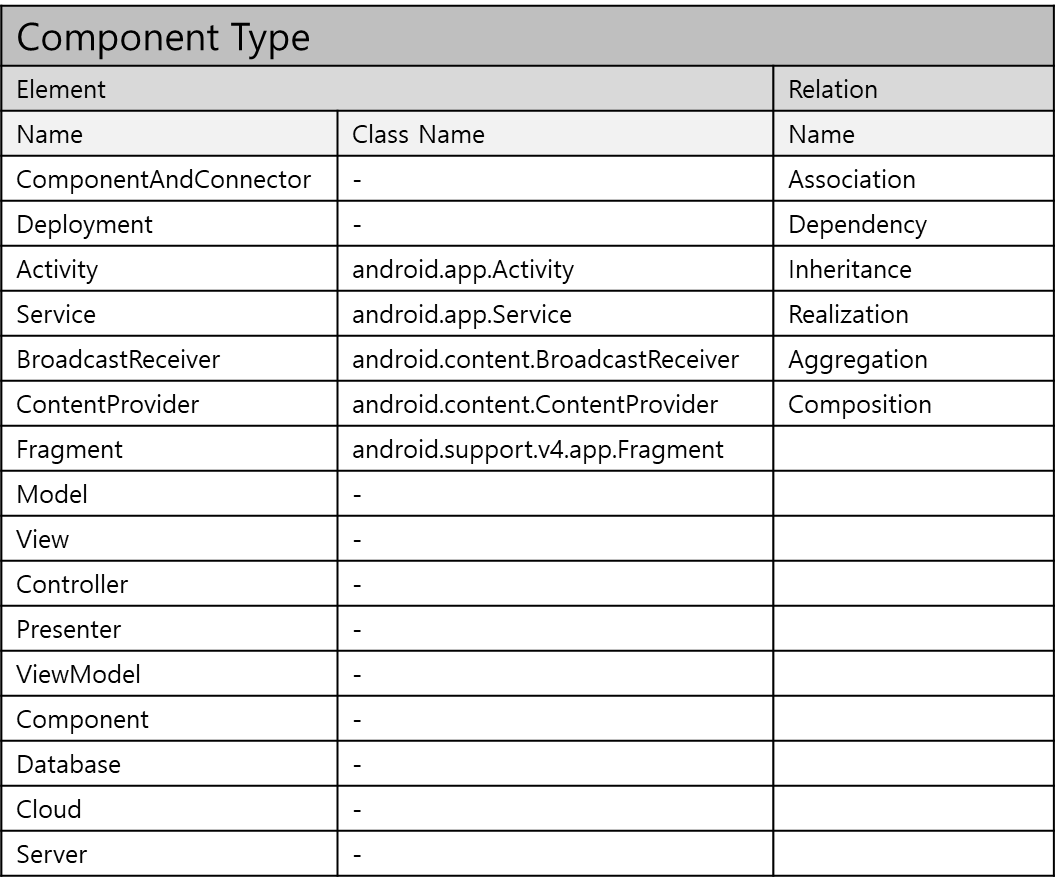


그림 . Component Type Table

Code Generation Library는 상기 Table을 기준으로 아래와 같은 package 및 Class를 생성한다.



* + 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 Name으로 package 구문을 추가하고
2. Component Type Table을 기준으로 각 Component에 해당하는 Class를 import하고
3. 현재 Element의 Name 및 Component Type을 ClassPrototype에 전달하여 Class 선언 문구를 받아오고
4. Relation의 to에 해당하는 name을 ClassProtytype에 전달하여 Member 변수 선언 문구를 받아와서 from에 해당하는 name의 Class에 추가한다.

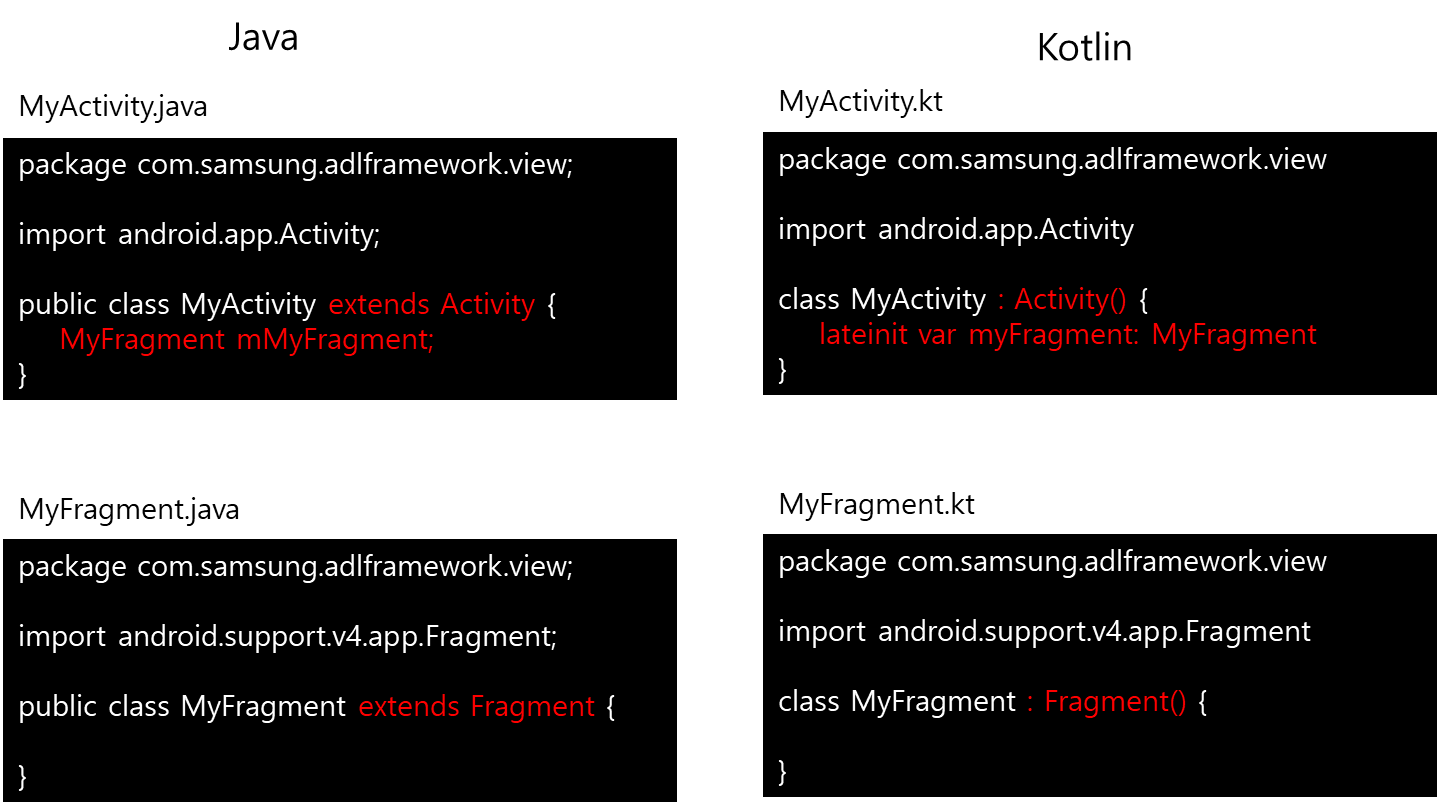


그림 . Code Generate: Java vs Kotlin

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

ADL Parsing중에는 유효하지 않은 Component Type을 기입하거나, 필수적인 name 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. 후보 구조 평가

본 장에서는 도출된 후보 구조를 사용편의성, 성능, 변경 용이성 세가지 관점에서 비교 평가를 진행한다.

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

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

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (++) 간결하게 ADL Code 작성할 수 있다. |
| QA\_02 ADL Code의 DOT Code 변환시간 | (-) DOT Code를 그대로 사용하는 경우에 비해 소요시간이 발생한다. |
| QA\_04 Component Type의 변경 용이성 | (+) Component Type을 그대로 사용하기 때문에 추가적 변환 시간이 발생하지 않는다. |

후보구조1.은 본 시스템의 차별화된 사용성을 만들 수 있는 구조로 채택한다. 성능 측정시에도 해당 후보구조 설명에 사용된 component가 7개 Relation이 3개 있는 구조를 표현했을 때

**0.69[간결성] = 599[ADL Code 문자 수] / 867[변환된 DOT 코드 문자 수]**

간결성이 매우 우수하며, Element와 Relation으로만 구성되어 있어서 새로운 ADL 언어 학습에도 어려움이 없을 것으로 판단한다. DOT 변환시간이 발생하기에 DOT를 그대로 ADL로 사용하는 것 대비 변환시간이 길어지는 Risk가 존재하지만, DOT는 단지 Graph를 그리기 위한 language일 뿐 Architecture를 드러내는 ADL로 사용하기에는 직관적이지 않아 적합하지 않다.

**후보구조 2. ADL을 따로 정의하지 않고 기존 ADL을 사용**(채택하지 않음)

본 시스템의 목적이 쉽고 편하게 Architecture를 작성하는 것이며 기존 ADL은 문법이 복잡해서 학습이 어렵다. 또한 기존 ADL Tool과의 차별성을 갖기 어렵기 때문에 채택하지 않는다.

**후보구조 3. 기존 Markup language의 문법을 그대로 사용**(채택하지 않음)

HTML, XML과 같은 Markup language는 익숙한 사용자에게는 별도로 학습이 필요하지 않고, 본 시스템에서 정의한 Component Type을 그대로 사용할 수 있는 강점이 있으나, 후보구조 1.과 마찬가지로 DOT로의 Parsing 작업이 필요하며 간결성 측면에서 약점이 있어서 채택하지 않는다.

* 1. QA\_02 ADL Code의 DOT Code 변환성능에 관련한 후보구조 평가

**후보구조 4. ADL을 DOT Language와 동일하게 정의** (채택하지 않음)

변환성능은 다른 구조에 비해 월등히 좋지만, Android에 특화된 본 시스템의 차별성을 가지기 어렵고 DOT 자체로는 원하는 Architecture를 그리기도 쉽지 않는 만큼 사용성이 떨어져서 채택하지 않는다.

**후보구조 5. Dot language Parser library를 사용** (채택하지 않음)

DOT parsing을 외부 Library에 의존하는 방법으로 Component Type 변경에 제약이 많고 성능도 보장할 수 없어서 채택하지 않는다.

**후보구조 6. ADL Code분석을 단계별로 나누어 동작** (채택함)

**후보구조 7. ADL Code 분석을 Pipe & filter 방식으로 수행** (채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (+) 시스템에서 자체적으로 Parsing을 진행해서 정의한 ADL 사용가능 |
| QA\_02 ADL Code의 DOT Code 변환시간 | (+) 각 단계를 각각의Thread에서 처리해서 성능 향상 |
| QA\_05 Code 생성 Language 확장성 | (+) Parse Tree를 생성하기 때문에 Language 확장성에 이점 |

후보구조 6.과 후보구조 7.은 ADL Code를 Parsing하는 방법을 단계별로 진행하게 되어 ADL 변경 용이성을 높일 수 있고, 구분된 단계를 Pipe & Filter 방식으로 병렬 Thread처리함으로써 변환 성능을 더욱 높일 수 있다. 또한 변환과정에서 만들어지는 Parse Tree는 그대로 Code 생성에 활용될 수 있어서 Code 생성 Language 확장성에도 이점이 있어서 채택한다.

**후보구조 8. 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.

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

**후보구조 9. 연속적으로 Graph를 갱신** (채택하지 않음)

Pipe & Filter 구조에 따라 각 Token마다 Graph를 갱신함에 따라 첫 Graph 반환 성능은 높겠으나, 너무 빈번한 Graph 생성으로 System Resource 점유가 높아져 DOT 변환 성능에 악영향을 끼치며 이에 따라 최종 Graph 생성에도 지장이 발생할 수 있어서 채택하지 않는다.

**후보구조 10. 일정한 주기로 Graph를 갱신** (채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_02 ADL Code의 DOT Code 변환시간 | (-) DOT Code 완성전에 Graph 반환 작업으로 인한 Resource 분배로 변환 성능에 악영향 |
| QA\_03 Translate 요청 시 첫 Graph 반환시간 | (++) Translate시에 즉각적으로 Graph반환이 가능 |

후보구조 9. 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 DOT 변환 성능에 끼치는 영향을 최소화했다. 이에 따라 최종 Graph 생성에도 큰 지연을 발생시키지 않는 구조로 평가되며 이에 따라 채택한다.

**후보구조 11. Translate 요청전에 Background parsing**(채택하지 않음)

본 구조를 적용하기 위해서는 ADL 도구에서 Code 변경이 이루어질 때마다 ADL Framework System으로 Background Parsing request를 전달해야 하며, 이는 불필요한 Background 작업을 반복적으로 수행하게 된다. 사용자가 원하는 시점에 Translate를 한번 진행해도 후보구조 10. 일정한 주기로 Graph를 갱신으로 충분한 즉시 반응성을 가지게 되며 이에 따라 채택하지 않는다.

* 1. QA\_04 Component Type 변경 용이성 후보구조 검토

**후보구조 12. 별도의 Component Type library로 발행**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_01 ADL Code 작성의 편의성 | (+) Component Type의 추가로 ADL Code 작성이 더 수월해짐 |
| QA\_02 ADL Code의 DOT Code 변환시간 | (-) Library linking 절차로 인해 다소간의 성능 저하 발생 |
| QA\_03 Translate 요청 시 첫 Graph 반환시간 | (-) Library linking 절차로 인해 다소간의 성능 저하 발생 |
| QA\_04 Component Type의 변경 용이성 | (++) ADL Framework의 어떠한 변경도 없이 Library 변경만으로 Component Type 변경 가능 |

Android의 다양한 Component들을 ADL 언어차원에서 지정할 수 있도록 정의한 Component Type을 library 교체만으로 변경할 수 있게 함으로써 Component Type의 변경 용이성을 극대화할 수 있다고 평가된다. Component Type을 변경하고자 할 때 ADL Framework 자체를 빌드 / 테스트 / 배포할 필요없이 Component Type에 대한 Library만 빌드 / 테스트 / 배포하면 됨에 따라 개발 및 발행의 효율성이 증대된다. Risk라 볼 수 있는 library linking 절차는 성능에 끼치는 영향이 미미하다고 판단하며 본 구조를 채택한다.

**후보구조 13. Decorate Pattern을 적용**(채택하지 않음)  
**후보구조 14. Strategy Pattern을 적용**(채택하지 않음)

두가지 구조 모두 손쉽게 Component Type을 변경할 수 있는 구조이며, 변경 시 기존 모듈의 영향이 적어서 변경 용이성이 높다고 볼 수 있으나, 변경된 Component Type을 발행하기 위해서는 ADL Framework 자체를 빌드 / 테스트 / 배포할 필요가 있어서 후보구조 12. 별도의 Component Type library로 발행에 비해 비용이 많이 발생한다고 판단되어 채택하지 않는다.

* 1. QA\_05 Code 생성 Language 확장성 후보구조 검토

**후보구조 15. Code Generation Library 적용**(채택하지 않음)

후보구조 12. 별도의 Component Type library로 발행처럼 별도의 Library로 Code 생성을 위임하는 형태여서 변경 용이성이 높다고 할 수 있으나, Code Generation에서 사용되는 Parse Tree의 구조가 변경된다면 오동작의 Risk가 있어서 별도의 Library에서 해당 동작을 하기 보다는 ADL Framework 내에서 Code Generation 동작을 수행하는 것이 안정적이라 볼 수 있다. 이에 따라 채택하지 않는다.

**후보구조 16. Language별 Class Prototype변경**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_05 Code 생성 Language 확장성 | (+) 다른 모듈에 영향이 거의 없게 변경이 가능 |

현재Android개발을 위한 Language로 Kotlin 사용빈도가 높아지고 있지만, 10년이상 Java가 거의 독점해왔다. 이는 Code 생성을 위한 Language 확장이 자주 발생하는 일은 아니라는 것을 반증하며, 이를 위해 굳이 Library 형태로 Parse Tree 구조 변경에 따른 오동작의 우려를 가지고 적용하기 보다는 ADL Framework 자체의 version update시에 적용을 하는 것이 안정적이다고 판단한다. 사용자가 필요에 따라 Language 설정을 변경해서 Code Generate를 하는 사용성에 가장 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 17. ADL 분석 단계에서 Error 발생시 Thread 종료**(채택함)

|  |  |
| --- | --- |
| 품질 요구사항 | 영 향 |
| QA\_02 ADL Code의 DOT Code 변환시간 | (+) Error 발생시 불필요한 변환과정이 바로 종료됨에 따라 DOT Code 변환작업을 바로 마치게 됨 |
| NFR\_01 ADL Parsing중 Error 발생시 회복 시간 | (+) Error 발생시 바로 Task를 Destroy하고 State를 변경해서 회복이 빠름 |

Syntax Tree생성 과정에서 ADL 문법에 맞지 않는 Token의 경우 Error Handler가 이를 인지하고 TaskManager를 통해 Task를 바로 종료할 수 있어서 Error 발생에 대한 회복 시간이 제약조건인 1000ms 이하로 이루어질 것으로 판단되어 채택한다.

* 1. 후보 구조 평가 결과

|  |  |
| --- | --- |
| 후보구조 | 채택 여부 |
| 후보구조 1. Element - Relation으로만 구성된 ADL | 채택함 |
| 후보구조 2. ADL을 따로 정의하지 않고 기존 ADL을 사용 | 채택하지 않음 |
| 후보구조 3. 기존 Markup language의 문법을 그대로 사용 | 채택하지 않음 |
| 후보구조 4. ADL을 DOT Language와 동일하게 정의 | 채택하지 않음 |
| 후보구조 5. Dot language Parser library를 사용 | 채택하지 않음 |
| 후보구조 6. ADL Code분석을 단계별로 나누어 동작 | 채택함 |
| 후보구조 7. ADL Code 분석을 Pipe & filter 방식으로 수행 | 채택함 |
| 후보구조 8. Update를 위해 Token 및 Parse Tree 유지 | 채택하지 않음 |
| 후보구조 9. 연속적으로 Graph를 갱신 | 채택하지 않음 |
| 후보구조 10. 일정한 주기로 Graph를 갱신 | 채택함 |
| 후보구조 11. Translate 요청전에 Background parsing | 채택하지 않음 |
| 후보구조 12. 별도의 Component Type library로 발행 | 채택함 |
| 후보구조 13. Decorate Pattern을 적용 | 채택하지 않음 |
| 후보구조 14. Strategy Pattern을 적용 | 채택하지 않음 |
| 후보구조 15. Code Generation Library 적용 | 채택하지 않음 |
| 후보구조 16. Language별 Class Prototype변경 | 채택함 |
| 후보구조 17. ADL 분석 단계에서 Error 발생시 Thread 종료 | 채택함 |

1. 최종 구조 설계

아래는 다양한 후보구조를 통합한 본 시스템의 최종구조이다.



그림 . 최종구조 Logical View

시스템을 크게 3가지 Layer로 구분한 3-Tier Layered Architecture를 적용해 추후 각 Layer별 변경 용이성을 높였다. ADL Tool과의 Interface를 담당하는 Interface Layer와 실제 Translate 및 Code Generate를 담당하는 Logic Layer, 이때 사용되는 Token 및 Tree를 포함하는 Data Layer로 구성된다.

* 1. 시스템의 변경용이성 측면

|  |
| --- |
| 후보구조 1. Element - Relation으로만 구성된 ADL  후보구조 12. 별도의 Component Type library로 발행  후보구조 16. Language별 Class Prototype변경 |

Element - Relation으로만 구성된 ADL은 Component Type의 변경만으로 언어에서 지원하는 Architecture의 Component들을 변경할 수 있다. 언어 차원에서 변경이 용이한 구조를 가지고 있으며 이 Component Type은 ADL Framework 시스템에 포함되지 않고 별도 Library로 발행된다. Component Type을 변경할 때 ADL Framework자체를 빌드 / 테스트 / 배포하는 작업이 불필요하며 단지 해당 Library에 대한 수정만으로 변경이 가능하다. ADL Framework에서는 Component Type에 대한 Interface를 가지고 실제 Component Type을 참조한다. Layered Architecture는 이 Component Type을 참조하는 Layer를 Logic Layer에만 한정하게 하여 변경에 대한 Risk를 최소화하였다. 또한 Code Generate시에도 Component Type을 참조해서 각 Type에 mapping되는 Class를 생성하며 이 또한 Logic Layer에 배치되었다.

* 1. 시스템의 Translate 성능 측면

|  |
| --- |
| 후보구조 6. ADL Code분석을 단계별로 나누어 동작  후보구조 7. ADL Code 분석을 Pipe & filter 방식으로 수행  후보구조 10. 일정한 주기로 Graph를 갱신 |

Translate 성능을 위해서 Lexical Analyzer – Syntax Analyzer - Semantic Analyzer – DOT Writer 단계를 두고 동작하도록 하였고 이를 Pipe – Filter 병렬 Thread로 동작하여 성능을 높였다. 또한 Translate 요청 시 즉각적인 Graph 생성 및 이에 대한 Overhead 최소화를 위해 Scheduled Queue를 적용해 일정한 주기를 가지고 Graph를 update하도록 하였다.

아래 최종 구조 Deployment View는 상기 구조를 실제 동작상의 Process – Thread 관점에서 표현하였다.



그림 . 최종구조 Deployment View

* 1. 시스템의 신뢰성 측면

|  |
| --- |
| 후보구조 17. 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.

* 1. 시스템의 Risk & Management

본 ADL Framework은 자체적으로 정의한 ADL을 기반으로 각 Component Type이 드러날 수 있게 DOT language로 변환한 후 Graphviz를 통해서 Graph를 반환한다. 기본적으로 Grape 생성을 위해서 DOT language와 Graphviz library에 dependency가 있으며, Open Source라고 하나 언제든 상업적인 목적으로 사용에 제약이 발생할 수 있다. 또한 Graphviz에서 제공되는 Graph 형태만 표현할 수 있어서, 다양한 사용자의 Needs에 부합하지 못할 수 있다.

이런 RISK를 해결하기 위해서는 ADL Code 분석 단계 중 Parse Tree 생성이후에 진행하는 filter, DOT Writer 동작과 Graphviz Manager를 추상화해서 전략에 따라 사용하는 Grape 변환 Language 및 Library를 변경하기 용이한 구조로 확장이 필요하며, 지속적인 Research를 통해서 우수한 Graph 변환 package로 교체하거나 직접 제작 제작하는 것을 고려할 필요가 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)