JKIICE Journal of the Korea Institute of Information and Communication Engineering

한국정보통신학회논문지 Vol. 26, No. 10: 1509~1516, Oct. 2022

UML 상태 다이어그램을 위한 코드 구현 모델의 생성 방법

김윤호*

A Method of Generating Code Implementation Model for UML State Diagrams

Yun-Ho Kim*

*Professor, Department of Computer Engineering, Andong National University, Andong, 36729 Korea

요 약

본 논문에서는 UML 상태 다이어그램으로부터 코드 구현 모델을 생성하는 방법을 제시한다. 첫째로 상태 다이어그램의 상태를 객체화하고, 상태 디자인 패턴에 기반하여 동작 메커니즘을 구조화한다. 다음으로 이 구조에 기반하여 컨텍스트 클래스와 이의 인터페이스 역할을 하는 추상 상태 클래스, 그리고 하위 구상 클래스의 구현 코드를 생성하는 규칙들을 설정한다. 이들 규칙들은 Java의 언어 구조에 기반해서, 상태 다이어그램의 상태들과 동작들에 대한 코드 구현 모델을 생성하게 된다. 한편, 코드의 자동 생성을 위해서는 코드 모델로부터 코드 생성을 위해 구조화된 정보가 필요하다. 따라서, 코드 생성을 위한 정보를 메타 클래스 모델과 메타 행위 모델 형태로 구조화하여 구축한다. 이 메타 정보들에 기반하여 상태 다이어그램으로부터 Java 프로그래밍 언어로의 자동 코드 생성을 위한 엔진의 구축이 가능하다. 또한 코드 모델 생성 엔진은 독자적으로 또는 UML 도구의 상태 다이어그램 기능을 지원하는 도구에 통합된 모듈로서 사용될 수 있다.

ABSTRACT

This paper presents a method to generate a model of the code implementation for UML state diagrams. First, it promotes the states of a state machine into objects, and then it structures the behavior model on the mechanism of a state diagram based on State design pattern. Then, it establishes the rules of generating the code implementation, and using the rules, the Java code mode is generated for the implementations of State Diagrams in Java syntax grammar. In addition, Structuring the information of the code model is necessary for generating Java code automatically. The meta information is composed of Meta-Class Model and Meta-Behavior Model, on which we could construct the automatic code generating engine for UML State Diagrams. The implementation model generation method presented in this paper could be used as a stand-alone engine, or included and integrated as a module in the UML tools.

키워드 : UML 다이어그램, 객체 지향 모델링, 객체 지향 설계, 자동 코드 생성, 소프트웨어 모델링

Keywords : UML diagrams, Object-oriented modeling, Object-oriented design, automatic code generation, Software modeling

Received 15 September 2022, **Revised** 25 September 2022, **Accepted** 4 October 2022

*** Corresponding Author** Yun-Ho Kim(E-mail:javian99@gmail.com, Tel:+82-54-820-5898)
Professor, Department of Computer Engineering, Andong National University, Andong, 36729 Korea

Open Access http://doi.org/10.6109/jkiice.2022.26.10.1509 print ISSN: 2234-4772 online ISSN: 2288-4165

Ⅰ. 서 론

소프트웨어 시스템의 설계로부터 자동 코드 생성의 문제는 UML에 기반한 소프트웨어 개발 도구에 대한 연구에 있어서 많은 주목을 받아 왔으며, 다양한 분야에서 활발한 연구가 진행 중이다. UML의 상태 다이어그램은 객체지향 소프트웨어 개발에서 상호작용적 시스템을 설계하기 위해서 제안된 것이며, 상태 변화 주도적인 시스템에서 행위에 기반한 설계가 필요한 애플리케이션 개발에서 널리 사용되고 있다. UML 개발 도구 또는 독립적인 동작을 전제로 한 상태 다이어그램으로부터의 코드 구현에 대한 여러 연구들이 제시되어 왔다 [1]. Ali[2]에서는 상태 다이어그램의 실행 가능한 코드를 위한 모델을 제시하였으나 체계적인 모델 생성 과정에 대해서는 명료하게 제시되어 있지 않다 [3]. 상태 다이어그램으로부터 직접 실행가능 코드의 생성에 대한 연구들[4], 모델 주도의 코드 구현을 제시한 연구들 등이 이루어져 왔다 [5,6].

본 논문에서는 UML 도구에서 적용될 상태 다이어그램의 코드 구현을 위한 코드 모델을 정의하고, 이를 생성하기 위한 코드 모델 규칙들을 제시한다. 생성된 코드 모델을 기반으로 Java 코드로의 자동 생성을 하기 위해 사용될 코드 모델 정보를 메타 클래스 모델과 메타 행위 모델 형태로 구축한다. 이러한 코드 모델 정보 레포지토리는 코드를 자동 생성하는 엔진을 구현하는 데 사용될 수 있으며, 이 엔진은 단독으로 또는 설계 도구의 모듈으로서 통합 구현에 활용할 수 있다.

Ⅱ. 상태와 상태 클래스

상태 다이어그램의 실행 가능한 코드로의 구현을 위해서는 프로그래밍 언어로의 변환이 필요한데, 객체 지향 프로그래밍 언어를 사용하여 코드를 구현하는 데는 어려운 점이 많다. 그 이유는 대부분의 널리 사용되고 있는 객체 지향 프로그래밍 언어에서 상태 다이어그램에 대하여 구문적으로 직접적으로 지원하는 기능이 없기 때문이다. 상태 다이어그램을 객체지향언어인 Java 언어로 구현하기 위해서는 먼저 상태 다이어그램의 상태를 객체로 승격 (promote)시키는 것이 필요하다. 상태 다이어그램에서 상태는 상태 객체로 승격시켜 클래스로 매핑하고, 상태 다이어그램에서 표현된 행위는 메소드로 매핑해서 구현할 수 있다. 또한, 상태들의 이벤트나 액션 역시 메소드로 매핑하여 구현할 수 있다 [7]. 그리고, Java의 메소드는 UML의 오퍼레이션과 그 오퍼레이션의 실현과 대응된다. 오퍼레이션 실현은 시퀀스 다이어그램으로 표현하여 메소드 바디로 대응시킬 수 있다. 표 1은 본 논문에서 상태 다이어그램과 Java 언어 사이의 변환 관계를 보인다.

Table. 1 State Diagram (UML) to Java Transformation

UML	Java
State	Class
Event	Method
Action	Method
Entry/Exit	Method
Operation	Method header
Sequence Diagram	Method body

상태 다이어그램의 전형적인 예로서 두개의 상태 Stop과 Play를 가지는 뮤직 플레이어를 그림1에서 보인다. 초기 상태는 Stop이며, playBtn 이벤트가 발생하면 startPlay 액션을 실행하고 Play 상태로 전이한다. Play 상태에서 StopBtn 이벤트가 발생할 때, stopPlay 액션을 수행하고 Stop 상태로 전이한다.

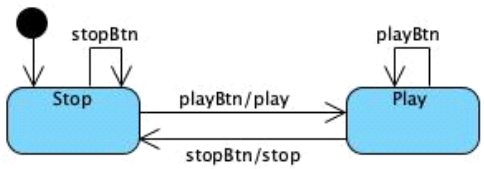

Fig. 1 The state diagram of Music Player.

상태 다이어그램의 구조적인 객체화를 위하여, 클래스들의 구조와 동작 메커니즘을 '상태 디자인 패턴'[8]에 기반하여 구현한다. 상태 디자인 패턴은 클라이언트의 관심에 부합하는 컨텍스트 (Context) 클래스와 이 클래스의 인터페이스 역할을 하는 추상 클래스 (State) 그리고 State 클래스를 상속하고 실현하는 구상 하위클래스 (concrete subclass)들로 구성된다.

그림 2는 뮤직 플레이어 예에 대한 상태 디자인 패턴에 기반하여 설계한 클래스 구조를 보인다. 클라이언트가 접근하는 MusicPlayer 클래스는 상태 디자인 패턴에서의 컨텍스트 (Context) 클래스에 해당하고, AbState

클래스는 State 클래스에 해당하며, Play와 Stop 클래스는 하위 구상클래스에 해당한다. 컨텍스트 클래스는 상태 전이를 추상 클래스에 위임하며, 상태 구상클래스들은 이벤트에 적절한 액션과 상태 전이를 실제로 행하도록 구상화된다.

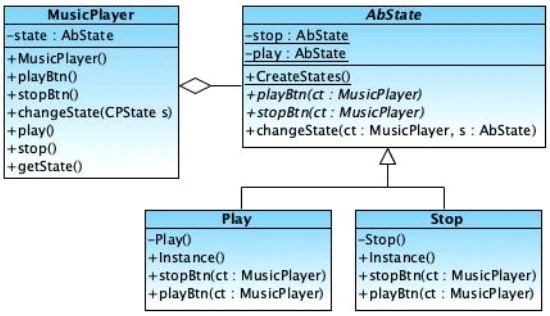

Fig. 2 Class Structure of Music Player.

Ⅲ. 코드 구현 모델

이 장에서는 상태 다이어그램으로부터 그 의미에 부합하는 코드로 구현하기 위한 코드 모델을 기술한다. 이 코드 모델에 기반하여 코드 생성 정보를 보유하는 메타 정보 모델을 생성하게 된다.

3.1. 컨텍스트 클래스의 코드 모델

컨텍스트 클래스의 코드 모델은 생성 규칙들에 의해서 템플릿 코드형태로 생성된다. Java언어를 전제로 하지만, 다른 언어에 대해서도 간단한 적합화(adaptation)로써 이들 규칙들이 적용 가능하다.

CT Rule 1: (Context templete)

```
public class Context {
}
```

CT 규칙 1은 컨텍스트 클래스의 템플릿 코드를 생성하는 규칙이다. 컨텍스트 클래스의 이름은 컨텍스트 상태의 이름과 대응되며, 기술의 편의상 컨텍스트 클래스의 이름을 Context로 한다.

CT rule 2: (current state)

```
private AbState state = state_name.Instance();
```

CT 규칙 2는 컨텍스트 클래스의 현재 상태 보관 변수를 처리하는 규칙이다. 컨텍스트 객체는 자신의 현재 상태에 대한 정보를 제공할 수 있어야 하므로, 현재 상태를 저장하는 변수를 가지고 있도록 구현한다. 여기서는 변수명을 `state'로 지정하며, 모든 개별 상태 객체의 타입을 지원해야 하므로 AbState 타입의 변수로 선언한다. 초기 상태의 정보로부터 싱글톤 객체로 초기화한다 (3.3절의 SC rule 2 참조). state_name은 상태 클래스의 이름을 나타내며, 현재 상태의 단일성을 지원하기 위해 전용(private) 변수로 한다. Ali 등 [3,4,6]에서는 전용으로 처리하지 않아 객체 중복 생성의 부담을 안고 있다.

CT rule 3: (constructor)

```
public class_name() {
        AbState.CreateStates();
}
```

CT 규칙 3은 컨텍스트 클래스의 생성자를 처리하는 규칙이다. 생성자에서 컨텍스트가 가질 수 있는 모든 상태 객체들의 집합을 생성하도록 하며, AbState에 위임한다 (3.2절의 AS rule 3 참조).

CT rule 4: (events)

```
public void event_name( ) {
        state.event_name(this);
}
```

CT 규칙 4는 컨텍스트 클래스의 이벤트들을 처리하는 메소드를 처리하는 규칙이다. 이벤트 처리는 개별 상태에서 직접 처리되는 것이 바람직하므로 이를 구체적 상태 클래스에 위임한다. state는 해당 상태를, event_name은 이벤트 메소드명을 의미한다. 또한 이벤트가 발생한 컨텍스트 정보 (this)를 매개변수로 하여 전달한다. 모든 이벤트들에 대해서 메소드 코드를 생성하여 컨텍스트 클래스 템플릿에 추가한다.

CT rule 5: (transition)

```
void changeState(AbState st) {
        state = st;
}
```

CT 규칙 5는 컨텍스트 클래스의 상태 전이에 대한 메소드를 처리하는 규칙이다. 컨텍스트에서 상태 전이는 현재 상태를 전이할 상태로 변경시키는 것으로 이루어진다. 변경할 상태 st를 매개변수로 받아 전이된다.

CT rule 6: (actions)

```
void action_name( ) {
}
```

CT 규칙 6은 컨텍스트 클래스의 액션들을 처리하는

메소드를 처리하는 규칙이다. 상태들에 대한 액션은 컨텍스트 클래스의 고유한 기능이므로 컨텍스트 클래스 내에 메소드를 구현한다.

3.2. 상태 추상클래스의 코드 모델

상태 추상클래스에 대한 코드 구현 모델의 생성은 상태 추상클래스의 템플릿 코드를 생성하는 것으로 시작한다.

AS rule 1: (abstract state template)

```
abstact class AbState {
}
```

AS 규칙 1은 상태 추상 클래스 AbState의 템플릿 코드를 생성하는 규칙이다. 여기서, 상태 추상클래스명을 컨텍스트 종류에 따라 다른 이름으로 부여할 수도 있으나, 하나의 이름으로 공통되게 사용하는 것이 가능하다. 따라서, 상태 추상클래스의 이름을 AbState로 지정하여 구현한다.

AS rule 2: (state objects)

```
static  AbState @state_name;
```

AC 규칙 2는 AbState 클래스의 모든 서로 다른 하위 구상 상태 객체들을 저장하는 변수들을 처리하는 규칙이다. 미리 정의된 상태들의 집합은 런타임에서 변경되지 않으므로, 상태 객체들은 한 번 생성하여 공유하여 사용하는 것이 효율적이다. 따라서 이들 변수들을 정적 변수로 로 하여 공유 사용으로 인한 효율성을 확보한다. 상태 저장 변수명 @state_name은 해당 상태의 이름 state_name을 상징하는 변수명으로 설정한다.

AS rule 3: (create states)

```
static void CreateStates( ) {
     while(isState()) {
          @state_name = state_name.Instance( ); }
}
```

AS 규칙 3은 AbState 클래스의 모든 하위 구상 상태 객체들의 생성을 처리하는 규칙이다. CreateState() 메소드는 모든 구상 상태 클래스로부터의 객체들을 모두 생성하여, 상태 객체 보관 변수들에 저장해두도록 구현한다. Instance() 메소드는 개별 상태 객체를 싱글톤으로 생성하는 메소드이며, 컨텍스트로부터 객체 생성 책임을 전적으로 위임받을 수 있도록 정적 메소드로 한다.

AS rule 4: (abstract events)

```
public abstract void event_name (class_name ct);
```

AS 규칙 4는 AbState 클래스의 이벤트 메소드들을 추상화 처리하는 규칙이다. 상태 추상클래스 AbState 구상 상태 클래스들에서 처리해야할 모든 이벤트 처리 메소드들을 추상 메소드로 정의한다. 이벤트가 발생한 컨텍스트 객체 ct를 매개변수로 한다.

AS rule 5: (transition)

```
void changeState(class_name ct, state_name st) {
          ct.changeState(st);
}
```

AS 규칙 5는 AbState 클래스의 상태 전이를 처리하는 규칙이다. changeState()는 어느 컨텍스트 객체가 어떤 상태로 전이할 것인지를 구현한다. 컨텍스트 객체 ct의 changeState()에 전이할 상태 객체 st를 매개변수로 주어 컨텍스트 객체의 상태 변경이 실제로 이루어지도록 한다.

3.3. 하위 구상클래스의 코드 모델

AbState 클래스를 상속하는 하위 구상 클래스의 코드 모델 생성은 AbState 클래스를 구현하는 구상 클래스들의 템플릿 코드를 생성하는 것으로 시작한다.

CS rule 1: (concrete state template)

```
class state_name extends AbState {
}
```

CS rule 1은 구상 상태 클래스의 각각의 클래스 템플릿을 생성하는 규칙이다. AbState 클래스의 상속을 클래스 헤더에 추가하여 템플릿을 생성한다.

SC rule 2: (Instance)

```
static AbState Instance() {
     if ( @state_name == null)  {
          @state_name = new state_name();
     };
     return @state_name;
}
```

CS rule 2는 상태 구상클래스의 상태 객체를 생성하는 규칙이다. 상태 객체들의 집합은 한 번 정의되면 런타임에서 변경이 되지 않으므로, 객체들은 한 번 생성되고, 이후 여러 컨텍스트 객체들이 공유하여 사용하게 하는 것이 중복 생성을 방지하여 효과적이다. 따라서, 구상 클래스의 객체들을 싱글톤으로 생성하도록 하며, 싱글톤 디자인 패턴 [8]에 기반하여 Instance() 메소드를 구현한다.

SC rule 3: (constructor)
private class_name {
}

CS rule 3은 Instance() 메소드에 의해서만 객체가 생성되도록 하는 생성자 억제 규칙이다. 앞서 CS rule 2의 Instance() 메소드에 의해서만 싱글톤 상태 객체를 생성하도록 하기 위해서는 기존의 생성자를 억제하여야 한다. 따라서, 생성자 전용(private)으로 구현하여 Instance() 메소드 내에서만 호출되도록 한다.

SC rule 4: (events)
public void event_name](Context ct) {
    // ct.@action_name();
    // changeState(ct,@state_name);
}

CS rule 4는 상태 구상클래스의 이벤트 메소드를 처리하는 규칙이다. 상태 구상 클래스의 이벤트 메소드는 추상 이벤트 메소드로 선언된 이벤트 메소드들을 모두 구현하는 메소드들이다. 이벤트 처리는 정의된 액션과 상태 전이를 수행하는 것으로 구성되며, 이벤트가 발생한 컨텍스트를 매개변수 ct로 받아오게 된다. 상태 다이어그램에서 정의된 해당 이벤트에서 수행할 액션 action() 메소드의 정보로부터 액션 수행 코드를 생성하며, 상태 전이 정보로부터 전이할 상태 @state_name를 changeState() 메소드에 매개변수로 추가하여 상태 전이 코드를 생성한다. 한편 특정 이벤트 메소드에서 어떠한 액션과 어떠한 상태로 전이해야 하는지에 대한 정보는 상태 전이의 행위 정보로부터 가져오게 되는 데, 이에 대해서는 다음의 4.2절의 메타행위를 논의하는 절에서 자세히 기술한다.

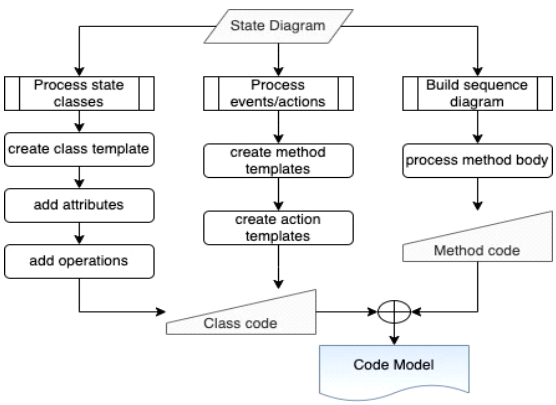

Fig. 3 Procedure of generating Code Model.

이 장에서 기술한 컨텍스트, 상태 추상, 하위 구상 클래스들에 대한 코드 모델을 생성하는 절차를 통합하여 보이면 다음의 그림 3과 같다.

Ⅳ. 메타 정보 모델

4.1. 메타 클래스 모델

앞 절에서 기술한 코드 모델로부터 Java 코드를 자동으로 생성하기 위해서는 클래스들에 대한 '클래스 상위 정보 (메타 클래스)'가 필요하다. 메타클래스 정보 구축은 Kim[9]에서 제시한 방법을 확장하여, 코드 구현을 위한 메타클래스 모델을 구축한다. 메타클래스의 기본 구조는 그림 4와 같으며, 각 클래스들 자체에 대한 상세한 상위 구조적 정보를 가지고 있다.

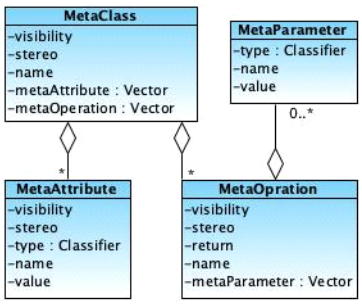

Fig. 4 The Structure & Relationship of Meta-Class.

메타 클래스의 전체적인 구조는 속성과 메소드 (또는 오퍼레이션)를 가지는 클래스의 구조와 본질적으로 유사하다. MetaClass는 MetaAttribute와 MetaOperation으로 구성되며, MetaOperation은 MetaParameter로 구서된다. MetaClass에서의 name 속성은 특정 클래스의 이름을 값으로 가지고, MetaAttribute의 name 속성은 특정 클래스의 속성의 이름을 값으로, 그리고 MetaOperation의 name 속성은 특정 클래스의 오퍼레이션 (메소드 헤더)의 이름을 갖는 등이다.

코드 모델을 입력으로 받아 메타클래스를 생성하는 절차를 보이면 그림 5와 같다. 메타클래스 생성기는 메타코드 정보제공자로부터 정보를 받아 코드 모델의 각 클래스들에 대한 메타클래스들을 생성한다. 각 메타 정보를 구축하는 세부 알고리즘은 Kim[9]에서의 알고리즘을 메타코드 모델을 처리할 수 있도록 확장하여 메타

클래스 생성기를 구현하였다.

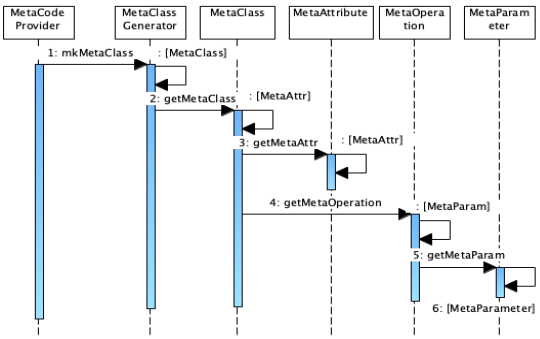

Fig. 5 Procedure of generating Meta-Class Object.

그림 6은 뮤직 플레이어 예의 컨텍스트 클래스 MusicPlayer 클래스에 대한 메타클래스 (CP)를 생성한 결과를 보여 준다. CP 메타클래스 객체를 중심으로 state 변수에 대한 메타속성, playBtn과 stopBtn 이벤트에 대한 메타오퍼레이션, 그리고 각 메타오퍼레이션에 대한 메타파라미터, 그리고 play, stop 액션에 대한 메타오퍼레이션과 메타파라미터 정보들이 메타클래스들로 구성되어 있다.

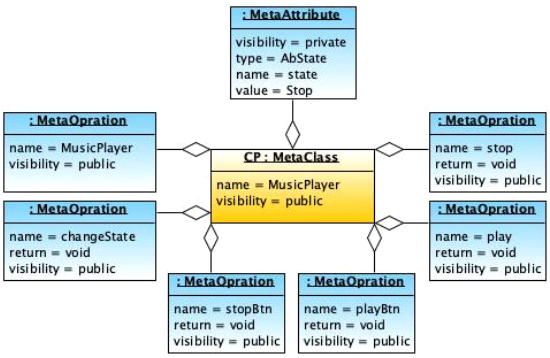

Fig. 6 Structure of Meta-Classes for Music Player.

4.2. 메타 행위 모델

메타클래스에는 클래스에 대한 상위 정보를 포함하고 있지만, 상태 다이어그램에서의 특정 이벤트에 대한 액션과 전이할 상태에 대한 정보는 가지고 있지 않다. 이러한 정보는 상태 전이 다이어그램으로부터 상태 정보를 가져오는 방법이 가장 효율적이다 [10]. 따라서, 상태 다이어그램으로부터 이벤트에 대한 상태 전이 정보를 분석하여 시퀀스 다이어그램으로 표현하며, 이 정보로부터 상태 전이 코드를 생성하는 방법을 사용하여 이벤트 메소드를 구현한다. 이렇게 하면 런타임에서 상태 집합이 변경될 경우에도 동적 자동 코드 생성 기법에 기반한 약간의 수정 작업으로 간단히 해결이 가능한 이점이 있다. 시퀀스 다이어그램으로부터 코드를 생성하는 방법 Kim[11,12]을 확장하여 상태 전이 메소드들을 생성하는 방법으로 사용한다.

앞서의 그림1의 상태 다이어그램에서 상태 전이 상황을 순차다이어그램으로 표현하면 그림 7과 같으며, 이에 대한 상태 전이 정보의 해석은 다음과 같다. alt 프레임은 가드 [state=Play]이 참인 경우에는 alt 프레임의 점선 상단부분이, 거짓인 경우에는 alt 프레임의 점선 하단부분이 수행된다.

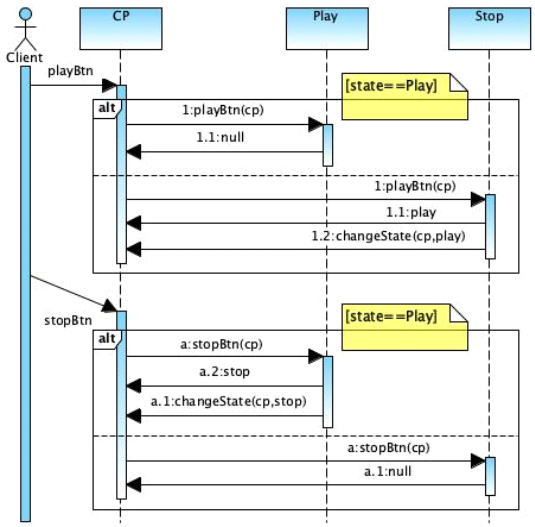

Fig. 7 Sequence Diagram of the Event Behaviors on the State Objects.

그림 7로부터 이벤트 처리 메소드의 바디 구현에 필요한 상태 전이 정보를 정리하면 다음의 표 2와 같다.

Table. 2 Action & transition information of events on a specific state.

State	Event	Action	Transition
Play	PlayBtn	play	Play
	stopBtn	stop	Stop
Stop	playBtn	play	Play
	stop	stop	Stop

각 상태에 대한 이벤트/전이에 관한 정보들로부터 Play 클래스에서 이벤트 메소드 playBtn(), stopBtn()의 바디 부분을 구현한 부분은 다음과 같다.

```
void playBtn(Context ct) {
        ct.play();
        changeState(ct,play);
}
void stopBtn(Context ct) {
        ct.stop();
        changeState(ct,stop);
}
```

이상에서 제시한 코드 모델 생성 규칙들과 정보 모델을 그림 1의 뮤직 플레이어에 적용한 최종적인 코드 모델의 결과를 보이면 그림8~그림10과 같다. Stop 클래스는 Play 클래스와 유사한 형태로 생성된다.

```
public class Context {
    private  state = state_Stop.Instance();
        Context() {
                AbState.CreateStates();
        }
        public void playBtn( ) {
                state.playBtn(this);
        }
        public void stopBtn() {
                state.stopBtn(this);
        }
        void changeState(AbState st) {
                state = st;
        void play( ) { }
        void stop( ) { }
}
```

Fig. 8 Code model for Context class.

```
abstract class AbState {
    static AbState play;;
    static AbState stop;
    static void CreateStates( ) {
        while(getState()) {
            play = play.Instance();
            stop = stop.Instance();
        }
    }
    public abstract void playBtn (Context ct);
    public abstract void stopBtn (Context ct);
    void changeState(Context ct, AbState st) {
        ct.changeState(st);
    }
}
```

Fig. 9 Code model for AbState class.

```
class Play extends  AbState  {
    private Play() {  }
    static AbState Instance() {
        if ( play == null)  {
            play = new Play();
        }
        return play;
    }
    void playBtn(Context ct) {
        ct.play();
        changeState(ct.play);
    }
    void stopBtn(Context ct) {
        ct.stop();
        changeState(ct.stop);
    }
}
```

Fig. 10 Code model for Play class.

Ⅴ. 결 론

본 논문에서는 UML 상태 다이어그램으로부터 코드 구현 모델을 생성하는 방법을 제시하였다. 이를 위하여 상태 다이어그램의 상태와 행위를 객체화시키는 작업을 하고 상태 디자인 패턴에 기반하여 동작 메커니즘을 구조화하였다. 이에 기반하여 코드 모델을 생성하기 위한 규칙들을 컨텍스트 클래스, 상태 추상 클래스, 하위 구상클래스에 대해서 설정하였다. 이들 규칙들을 적용하여 코드 구현 모델을 생성하며, 이후 자동 코드 생성을 위한 정보를 구축하기 위하여 메타클래스와 메타 행위모델을 제시하였다. 이들 정보에 기반하여 Java 코드로의 자동 생성이 가능하게 된다. 본 논문에서 제시한 코드 모델 방법은 UML 도구의 모듈로서 또는 단독의 애플리케이션의 구현에 활용될 수 있을 것으로 기대한다. 또한 코드 모델 생성 규칙의 확장성을 확보하기 위하여 컴포지트 상태와 컨커런트 상태를 지원하기 위한 연구가 진행 중에 있다.

ACKNOWLEDGEMENT

This work was supported by a Research Grant of Andong National University.

References

[1] M. I. Mukhtar and B. S. Galadanci, "Automatic code generation from UML diagrams: The-State-of-the-art," *Science World Journal*, vol. 13, no. 4, pp. 47-60, Feb. 2019.

[2] J. Ali and J. Tanaka, "Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams," *ACIS International Journal of Computer and Information Science*, vol. 2, no. 1, pp. 24-36, Mar. 2001.

[3] M. Ehl and M. Konersmann, "Model-based Monitoring of Integrated UML State Machine Models and Code," in *Proceedings of the Software Engineering 2021 (Satellite Events)*, Braunschweig/Virtual, Germany, pp. 12, 2021.

[4] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elaasar, H. Aljamaan, and M. A. Garzon, "Enhanced code generation from UML composite state machines," in *Proceedings of 2nd International Conference on Model-Driven Engineering and Software Development*, Lisbon, Portugal, pp. 235-245, 2014.

[5] S. Viswanathan and P. Samuel, "Translation of Behavioral Models to Java Code and Enhance with State Charts," *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 6, pp. 294-304, 2014.

[6] M. L. Alvarez, I. Sarachaga, A. Burgos, E. Estevez, and M. Marcos, "A Methodological Approach to Model-Driven Design and Development of Automation Systems," *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 1, pp. 67-79, Jan. 2018.

[7] V. C. Pham, A. Radermacher, S. Gerard, and S. Li, "Complete Code Generation from UML State Machine," in *Proceedings of 5th International Conference on Model-driven Engineering and Software Development*, Porto, Portugal, pp. 208-219, 2017.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, USA, 1996.

[9] Y. Kim, "Information Structuring of Diagram Repository for UML Diagrams," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 23, no. 12, pp. 1588-1595, Dec. 2019.

[10] S. E. Viswanathan and P. Samuel, "Automatic Code Generation from UML State Chart Diagrams," *IEEE Access*, vol. 7, pp. 8591-8608, Jan. 2019.

[11] Y. Kim, "A Design of Constructing Diagram Repository for UML Diagram Tools," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 24, no. 2, pp. 244－251, Feb. 2020.

[12] Y. Kim, "A Method of Automatic Code Generation for UML Sequence Diagrams Based on Message Patterns," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 24, no. 7, pp. 857-865, Jul. 2020.

김윤호(Yun-Ho Kim)

1983 경북대학교 전자공학과 학사
1993 경북대학교 컴퓨터공학과 석사
1997 경북대학교 컴퓨터공학과 박사
1997 – 현재 안동대학교 컴퓨터공학과 교수
※관심분야 : 객체지향시스템, 모바일 소프트웨어, 인공지능, 병렬처리