Mozart IDE

User Manual

VMS SOLUTIONS

본 문서는 고객사 전용으로 작성한 것으로 (주)브이엠에스 솔루션스의 사전 서면 동의 없이 고객사 외부에 회람, 인용, 사본 배포를 금합니다.

This material was prepared by VMS Solutions, solely for the use of our clients, and it is not be relied on by any third party without VMS Solutions written consent.

MOZART IDE

목차

Level 1	Level 2	Level 3
MOZART 개요	INSTALLATION	SYSTEM REQUIREMENTS
		MOZART Client 설치
		라이선스 인증 방법
		MOZART 라이선스 키 요청 방법
	GETTINGSTARTED	Hello World Example
		Simple ETL Example
MOZART IDE	OVERVIEW	MOZART IDE 개요
		Feature Extension 개념
		Execution 모듈 라이브러리
		User Interface
	MOZART PROJECT	MOZART 프로젝트 생성 방법
		프로젝트 파일 구성
		VXML 파일 구조
		Definition 프로파일링
	MODEL	Model 개요 및 생성 방법
		Arguments 관리
		Data Source 관리
		Data Item 관리
		다중 Model 프로젝트
	MY OBJECTS	My Objects 개요
		My Objects 사용 방법
		Constants 정의 방법
		Enum 정의 방법
		Data Mart 사용 방법

MY METHOD	My Method 사용자 가이드
MAIN MODULE	Main 모듈 개요 및 실행 구조
	Configuration
	Main 컨트롤
	FE Control 로직 개발 방법
	Data Preloading
SIMULATION MODULE	Simulation 개요
	Simulation 모듈 컨트롤 라이브러리
	Weight Factor Method 개발
	Custom Event 개발
	Filter Method 개발
	Simulation 통계 수집(Statistics)
PEGGING MODULE	Pegging 모듈 개요
	Rule 편집
	Stage 편집
	Pegging 컨트롤 라이브러리
	Pegging Modeler
CBS MODULE	Capacity Bucket Simulation 개념 및 개요
	CBS 모듈 클래스 설명
	CBS Solver
	CBS Agent

MOZART 개요

INSTALLATION

SYSTEM REQUIREMENTS

ℹ️ **Update Date : 2022.08.31**

OS - MOZART Client(MOZART IDE, STUDIO, MMC, WISE)

Windows	Support	Note
Windows 11	Supported	**Only Support Visual Studio 2017, 2019**
Windows 10	Supported	**Only Support Visual Studio 2013,2015,2017,2019**
Windows 7	Not Supported	

OS - MOZART Server

Windows Server	Support	Note
Windows Server 2022	Supported	
Windows Server 2019	Supported	
Windows Server 2016	Supported	
Windows Server 2012 R2	Supported	
Windows Server 2012	Supported	
Windows Server 2008 R2 SP1	Supported	
Windows Server 2008	Not Supported	

Development Environment

.Net Framework Version	Support	Note
.Net Framework 4.8	Supported	
.Net Framwork 4.7.2	Supported	
.Net Framwork 4.7.1	Supported	
.Net Framwork 4.7	Supported	
.Net Framwork 4.6.2	Supported	
.Net Framwork 4.6.1	Supported	**MOZART Library 사용 버전**
.Net Framwork 4.6	Not Supported	

Development Tool

Visual Studio	Support	Note
Visual Studio 2022	Not Supported	**Not Supported in .NET Framework 4.6.1**
Visual Studio 2019	Supported	
Visual Studio 2017	Supported	
Visual Studio 2015	Supported	**Microsoft's official support end date: 2025-10-14**
Visual Studio 2013	Supported	**Microsoft's official support end date: 2024-04-09**

Hardware Requirements

Name	요구사양
최소 사양	**1.8 Ghz Processor Dual-Core CPU 이상, 4 GB RAM 이상, 10 GB HDD 이상, 인터넷 접속 1280x720 이상 해상도 지원 비디오 카드**
권장 사양	**2.4 Ghz Processor Dual-Core CPU 이상, 16 GB RAM 이상, 20 GB SSD 이상, 인터넷 접속 1366x766 이상 해상도 지원 비디오 카드**

MOZART Client 설치

개요

본 문서는 **MOZART Client 설치 파일을 다운받는 방법**과 **제품 설치/업데이트/삭제하는 방법**을 설명합니다. MOZART 제품을 설치하는 방법은 1) MozartClientExecute.exe 실행 파일을 통한 설치 방법과 2) Mozart Client Installer를 이용한 설치 방법이 있습니다. 두 방법 중 제품 설치 및 버전 업데이트의 편리성을 위해 **1) MozartClientExecute.exe 실행 파일을 통해 제품 설치**하는 것을 권장합니다. 그 외 수동으로 제품을 설치하고자 하는 경우 **2) Mozart Client Installer를 이용한 제품 설치 방법**을 확인하시기 바랍니다.

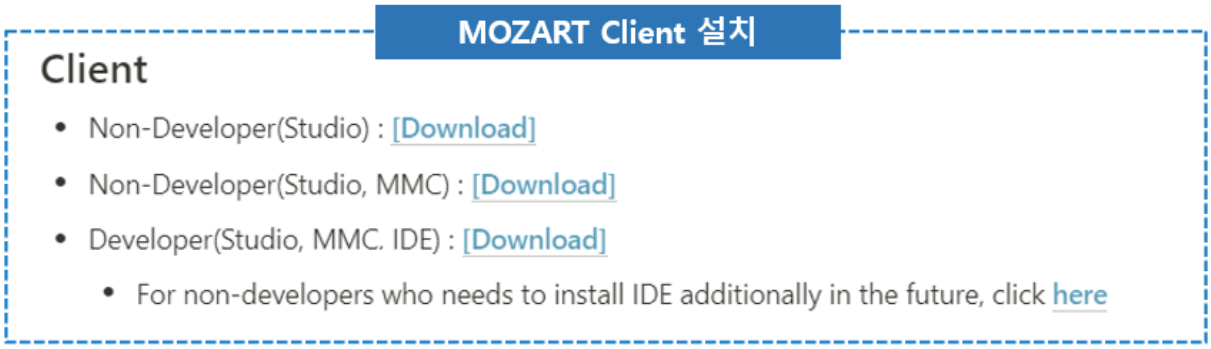
기능 사용에 필요한 스펙

- 적용버전: 2019.116.2.0 이상
- 적용대상: MOZART Client
- 적용 Domain: N/A

> 💡 **Note**
> 2019.115.1 버전 이하 사용자의 경우, 설치된 MOZART 제품을 모두 삭제 후 **Mozart Registry Cleaner**를 이용하여 완전 삭제 후 설치 및 업그레이드를 진행합니다.
> 자세한 내용은 Mozart Registry Cleaner Guide 문서를 참고하시기 바랍니다.

Client 설치 파일 다운로드

MOZART Client는 💿 MOZART Download Archive 에서 일반 사용자(Non-Developer)와 개발자(Developer)에 따라 **Client 제품을 다운로드** 받을 수 있습니다. 또한 필요에 따라 **도메인 라이브러리를 추가 다운로드** 받습니다.



도메인 라이브러리 설치

Domain Library

- Mozart Semicon Fab Domain Library: [Download]
- Mozart Semicon BE Domain Library: [Download]
- Mozart Semicon EDS Domain Library: [Download]
- Mozart LCD Domain Library: [Download]
- Mozart BCP Domain Library: [Download]
- Mozart General Domain Library: [Download]
- Mozart Basic Domain Library: [Download]

Mozart Download 페이지의 Client 파일과 Domain Library 파일

> 💡 2022.122.0 버전 부터 Product Download 페이지가 변경되었습니다.

MozartClientExecute.exe 사용 가이드

MozartClientExecute.exe 실행 파일을 통해 **Client** 제품을 자동 설치하는 방법을 안내합니다. 우선 아래 3가지 Client 파일의 구성을 확인하여 필요한 제품을 다운 받으시기 바랍니다.

Client 파일 구성

1. **Non-Developer(Studio)**

   MOZART Studio를 사용하는 **일반 사용자 대상** 설치 파일입니다.

   - 다운로드 파일 : **Studio.(version).zip**
   - 파일 구성 : Installer, ToBin, ToRoaming, Client.mdz, MozartClientExecute

2. **Non-Developer(Studio, MMC)**

   MOZART Studio, MMC(Mozart Management Console)를 사용하는 **일반 사용자 대상** 설치 파일입니다.

   - 다운로드 파일 : **Client.(version).zip**
   - 파일 구성 : Installer, ToBin, ToRoaming, Client.mdz, MMC.mdz, MozartClientExecute

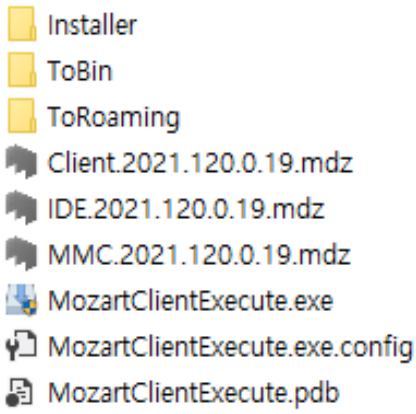
3. **Developer(Studio, MMC, IDE)**

MOZART Studio, MMC(Mozart Management Console), IDE를 사용하는 **개발자 대상** 설치 파일입니다.

- 다운로드 파일 : **Developer.(version).zip**
- 파일 구성 : Installer, ToBin, ToRoaming, Client.mdz, MMC.mdz, MozartClientExecute

▼ 설치 파일에 대한 자세한 설명

아래는 MOZART Client 파일에 구성되어 있는 설치 파일의 설명입니다.



ex) Developer(Studio, MMC, IDE) 파일 구성

📌 - **Installer** : MOZART Client Installer를 설치하는 msi 파일이 있는 폴더입니다.

- **ToBin** : MOZART 설치 파일을 재패키지 시, Mozart Client가 설치되는 폴더(*C:\Program Files (x86)\VMS\Mozart\Client\Bin*))에 복사할 파일을 추가하는 폴더 입니다.

- **ToRoaming** : MOZART 설치 파일을 재패키지 시, *%appdata%\Mozart*에 복사할Studio/IDE/MMC 관련 config 및 라이선스 파일을 추가하는 폴더입니다.

- **Client.(Version).mdz** : MOZART Studio를 설치하는 파일입니다.

- **IDE.(Version).mdz** : MOZART IDE를 설치하는 파일입니다.

- **MMC.(Version).mdz** : MMC(MOZART Management Console)를 설치하는 파일입니다.

- **MozartClientExecute.exe** : 해당 폴더에 존재하는 mdz 및 dll, config 파일을 자동 설치하는 파일입니다.

제품 파일 구성

Aa 제품	파일 구성
Client.mdz	Mozart Studio, 공용 Mozart DLL, DevExpress DLL
IDE.mdz	Visual Studio 확장 도구 인스톨러(vsix), Basic.vfe
MMC.mdz	MMC, MMC 관련 DLL
Domain Library.mdz	Domain DLL, Domain.vfe

2019.116.2 버전 이상 **MOZART Client** 설치 시 생성되는 파일의 경로는 아래와 같습니다.

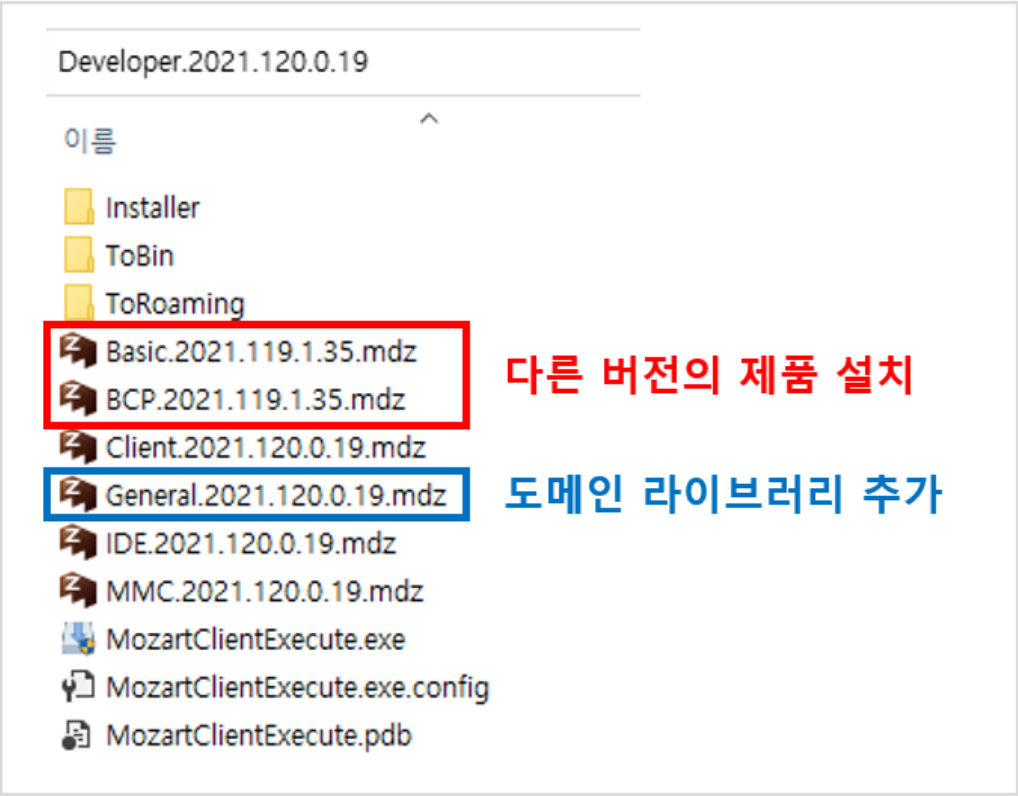
기본 설치 경로

Aa MDZ	설치 경로
Studio	Program Files\VMS\Client\Bin **설정 파일** : %appdata%\Mozart\XXX_Studio\voptions.config
MMC	Program Files\VMS\Client\Bin **설정 파일** : %appdata%\Mozart\Mozart Management Console 2\ServerExplorer.vmgr
IDE	**VS 2017, 2019 :** %Program Files (x86)%\Microsoft Visual Studio\[2017 or 2019]\[Edition: Professional, Enterprise]\Common7\IDE\Extensions\[랜덤 이름] **VS 2015 :** %Program Files (x86)%\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\[랜덤 이름] **VS 2013 :** %Program Files (x86)%\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\[랜덤 이름]
Domain Library	**DLL :** Program Files\VMS\Client\Bin **vfe :** ProgramData\Mozart\FeatureExtensions **vfe 압축 해제 :** %appdata%\Mozart\IDE\[Domain Library.vfe]
LinqPad	%localappdata%\LINQPad\Drivers\DriverContext\4.6\Mozart.LinqPad.Driver

> 💡 **Note**
> MOZART Client 기본 설치 경로는 Mozart Client Installer의 설치 경로가 기준이 됩니다. 기본 설치 경로를 변경하려는 경우 Installer 설치 시 경로를 변경하시기 바랍니다.

MozartClientExecute.exe 로 설치/업데이트 방법

1. 💿 MOZART Download Archive 에서 다운받은 Client 파일을 압축 해제합니다. 해당 폴더에서 **도메인 라이브러리를 추가 설치**하거나 설치된 **제품의 버전 변경(업그레이드, 다운그레이드)**하고자 하는 경우 아래 예시처럼 **mdz 파일을 추가**합니다.
   - 변경하고자 하는 버전의 mdz 파일 추가 (설치된 제품의 업그레이드, 다운그레이드)
   - Domain Library mdz 파일 추가



Developer 폴더에 도메인과 다른 버전의 제품을 추가한 예제

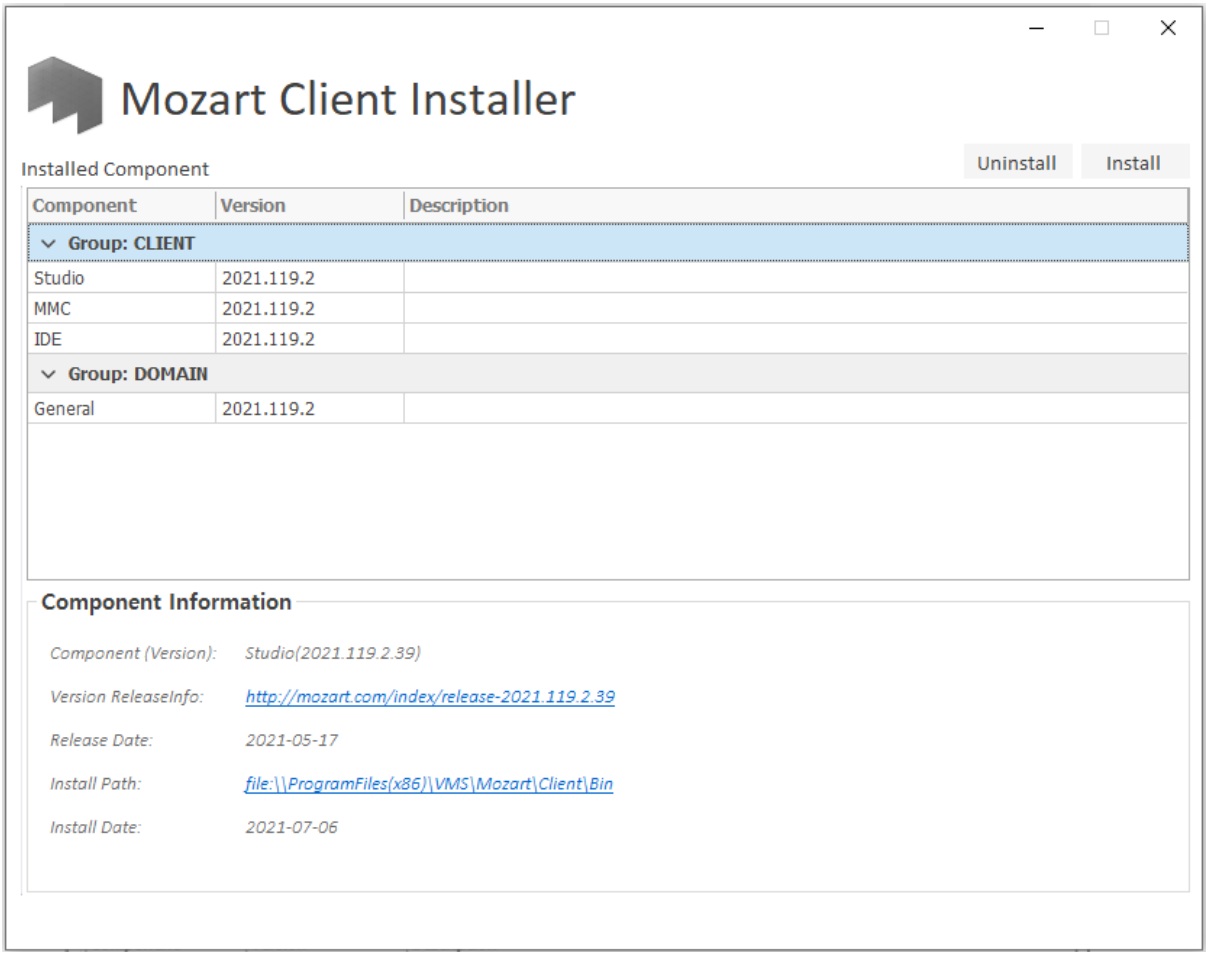
2. 추가적인 DLL, Config 파일도 함께 설치하고자 하는 경우, **ToBin**(*Client\Bin* 경로에 설치), **ToRoaming**(*%appdata%\Mozart* 경로에 설치) 폴더에 파일을 추가합니다. 자세한 방법은 설치 파일 재배포 시나리오 를 참고하시기 바랍니다.
3. 설치 파일이 모두 준비되었으면 해당 폴더의 **MozartClientExecute.exe** 실행 파일을 관리자 권한으로 실행하여 설치를 진행합니다.
4. **IDE**를 설치하는 경우 **VSIX Installer**가 자동 실행됩니다. Visual Studio Extension을 설치하기 위해 **[Install]** 버튼을 누릅니다.
5. 설치가 완료되면 **Mozart Client Installer**를 실행하여 제품 설치가 완료된 것을 확인합니다.

> ⚠️ **Note**
> IDE 제품은 MozartClientExecute.exe 실행 파일을 통한 버전 다운그레이드 설치는 불가합니다. IDE 제품은 Mozart Client Installer에서 삭제 후 낮은 버전을 설치하시기 바랍니다.

Mozart Client Installer 사용 가이드

Mozart Client Installer는 MOZART Client 및 Domain Library를 설치 및 삭제 할 수 있으며 설치된 제품의 빌드 버전 및 설치 경로와 같은 추가 정보를 확인 할 수 있습니다.

Mozart Client Installer 구성



Installed Component

Client 및 Domain Library 파일 별로 설치된 제품 컴포넌트를 버전과 함께 확인 할 수 있는 영역입니다. 우측 상단의 **[Uninstall], [Install]** 버튼을 통해 제품 설치 및 삭제가 가능합니다.

Component Information

Installed Component 에서 선택한 제품의 정보를 확인 할 수 있는 영역입니다.

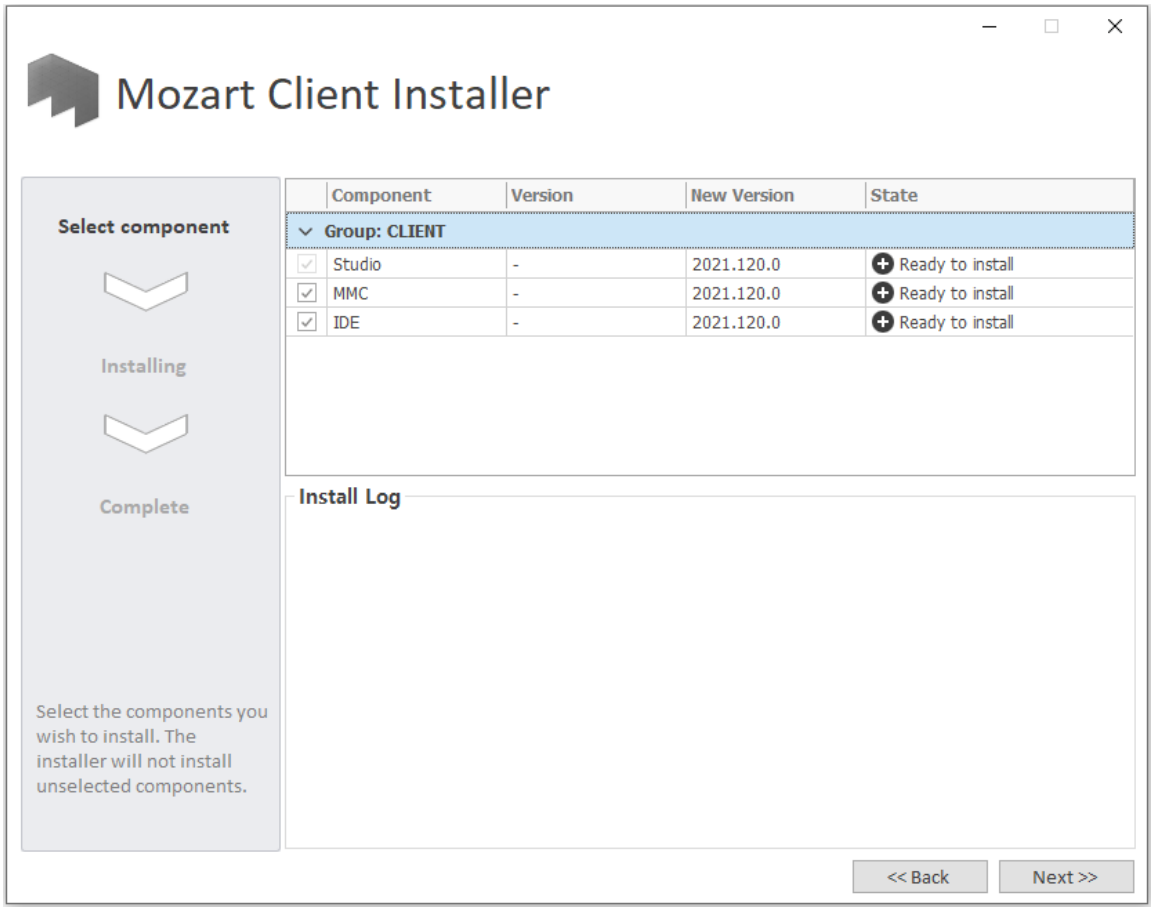
- **Component (Version)** : 선택한 제품 이름(설치 버전)
- **Version ReleaseInfo** : 릴리즈 목록을 확인할 수 있는 링크
- **Release Date** : 제품 배포일

- **Install Path** : 사용자 PC에 설치된 폴더 위치
- **Install Date** : 사용자 PC에 설치된 날짜

Mozart Client Installer 설치 및 MDZ 설치/업데이트 방법

Mozart Client Installer를 통해 제품을 수동으로 설치하는 것이 가능합니다. 그러나 Mozart Client Installer의 경우 설치된 제품보다 낮은 버전의 제품 설치는 불가능하여 버전 다운그레이드를 진행하려면 제품을 삭제한 후 낮은 버전을 설치해야 하는 제약이 있습니다. 제품의 버전 변경은 위의 MozartClientExecute.exe 실행 파일을 통한 설치를 참고하시기 바랍니다.

1. 💿 MOZART Download Archive 의 버전 페이지에서 필요한 제품을 확인하여 **Client 파일** 및 **도메인 라이브러리 파일**을 다운로드 합니다. (ex. *Developer.2021.120.0.19.zip, General.2021.120.0.19.mdz*)
2. 다운받은 Client 파일은 압축 해제 후 Installer 폴더 안의 ***MozartClientInstaller.Setup.msi*** 를 실행하여 **Mozart Client Installer**를 설치한 후 실행합니다. (설치경로 : *C:\Program Files (x86)\VMS\Mozart\Client\Bin*)
3. **Mozart Client Installer**에서 우측상단 **[Install]** 버튼을 클릭합니다.
4. 2번의 파일에서 설치 또는 버전 업데이트 할 제품 mdz 파일을 선택하여 [열기]를 클릭합니다. 이때 제품 중 **Studio(Client.mdz)는 필수 선택 또는 선행 설치** 되어야 다른 제품 설치가 가능합니다.



5. 설치할 제품을 선택한 후 **[next]** 버튼을 눌러 설치를 시작합니다.
6. **IDE**를 설치하는 경우 **VSIX Installer**가 자동 실행이 됩니다. Visual Studio Extension을 설치하기 위해 **[Install]** 버튼을 누릅니다.
7. 설치가 완료되면 **[Done]** 버튼을 눌러 설치를 마칩니다.
8. 필요에 따라 위 방법대로 도메인 라이브러리를 추가 설치합니다. (ex. *General.2021.120.0.19.mdz*)

⚠️ **Warning**

- **Studio(Client.mdz) 제품이 우선 설치**되어야 다른 제품 컴포넌트 설치가 가능합니다.

- Mozart Client Installer에서 **Studio 버전보다 낮은 버전의 다른 제품 설치는 불가능**합니다. (ex. Studio 2021.120.0 설치 후, IDE 2021.119.2 설치)
해당 경우는 Studio 삭제를 진행하는 대신 **MozartClientExecute.exe 실행 파일을 통해 버전이 낮은 제품 설치가 가능**합니다. 위의 설명을 참고하여 Studio 버전보다 낮은 제품의 mdz 파일을 MozartClientExecute.exe 실행 파일과 동일한 경로에 복사하여 설치를 진행하시기 바랍니다.

- 이미 설치된 **제품의 버전 업그레이드 설치는 가능**하나 설치된 제품보다 낮은 버전의 제품 설치는 불가능합니다. (ex. IDE 2021.120.0 설치 후, IDE 2021.119.2 설치)
제품의 버전을 낮추려면 설치된 제품을 삭제하여 낮은 버전을 설치하시기 바랍니다.

⚠️ 2022.122.0 버전부터 ML 라이브러리는 설치된 도메인 라이브러리와 동일한 버전만 설치 할 수 있도록 변경되었습니다. ML 라이브러리 설치 시 함께 사용하는 도메인 라이브러리를 먼저 설치하여 주시고, 동일한 버전의 ML 라이브러리 파일을 설치하시기 바랍니다.

제품 및 Installer 삭제 방법

MOZART 제품을 삭제하기 위해서는 **Mozart Client Installer**를 통해서 삭제합니다.

1. 실행 중인 MOZART 제품을 모두 종료합니다.
2. **Mozart Client Installer**를 실행합니다.
3. **[Uninstall]** 버튼을 누릅니다.
4. 삭제하려는 제품을 선택하고 **[Next]** 버튼을 누릅니다.

5. **IDE**를 삭제하는 경우 **VSIX Installer**가 실행되어 Visual Studio Extension 삭제가 진행됩니다.
6. 선택한 제품의 삭제가 모두 완료되면 **[Done]** 버튼을 누릅니다.
7. **Mozart Client Installer**를 삭제하려면 위의 방법대로 모든 제품을 삭제한 후 `제어판 > 프로그램 추가 및 제거`에서 **Mozart Client Installer**를 제거합니다.

> **Note**
> - Studio의 경우 기본 설치 프로그램으로 Studio만 삭제 할 수 없습니다. Studio를 삭제할 경우 설치되어 있는 모든 제품이 함께 삭제됩니다.
> - Mozart Client Installer를 삭제하여도 설치한 제품을 **[Uninstall]** 하지 않을 시 제품은 삭제되지 않고 남아있습니다.

라이선스 인증 방법

개요

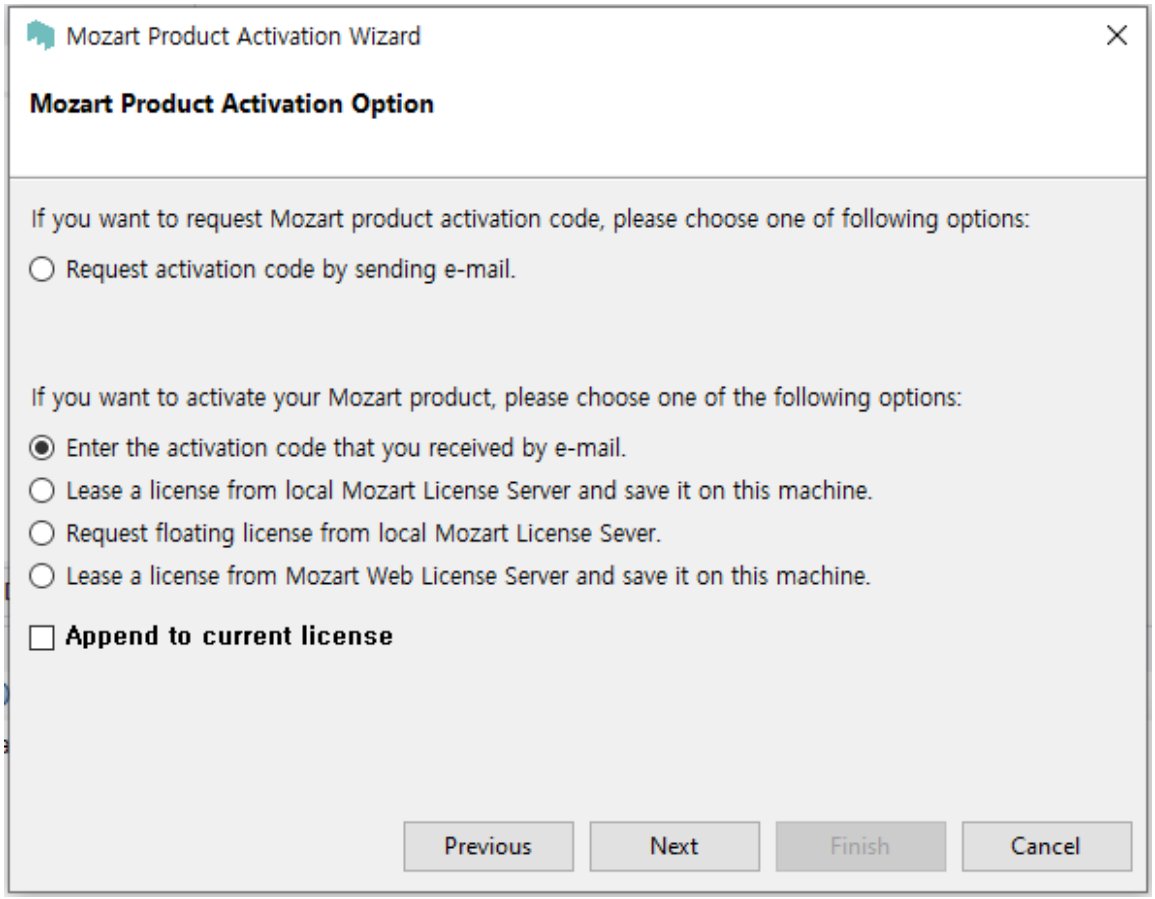
MOZART Studio 나 IDE 에서 라이선스가 인증되지 않은 상태에서는 **Mozart Product Activation Wizard** 창이 활성화되어 라이선스 인증 절차를 수행하도록 지원합니다. 현재 MOZART 에서 지원하는 라이선스 활성화 방식은 아래의 세 가지 방식입니다.

1. **Activation 코드 입력을 통한 라이선스 활성화** : 라이선스 키 요청을 통해 발급 받은 Activation 코드를 Activation Wizard 에 입력하여 활성화 하는 방식 입니다. 라이선스 발급 요청이 필요할 시에는 MOZART 라이선스 키 요청 방법을 참고하시기 바랍니다.
2. **Local License Service 를 통한 라이선스 활성화** : 사용자 조직 내부에 MOZART SERVER 를 설치한 경우 MOZART IDE, STUDIO 에 대한 라이선스를 자동 발행 할 수 있습니다. MOZART SERVER 는 Activation code 발급을 통해서만 활성화가 가능합니다.
3. **Floating License를 통한 라이선스 활성화**: 사용자 조직 내부에 MOZART SERVER를 설치한 경우 MOZART STUDIO에 대한 라이선스를 MOZART STUDIO를 매번 실행 할

때마다 서버를 통해 인증 받는 방식입니다. 서버와의 접속이 끊기거나 허용된 네트워크를 벗어나면 MOZART STUDIO 사용이 자동 중단 됩니다.

Activation code 입력을 통한 Activation 방법

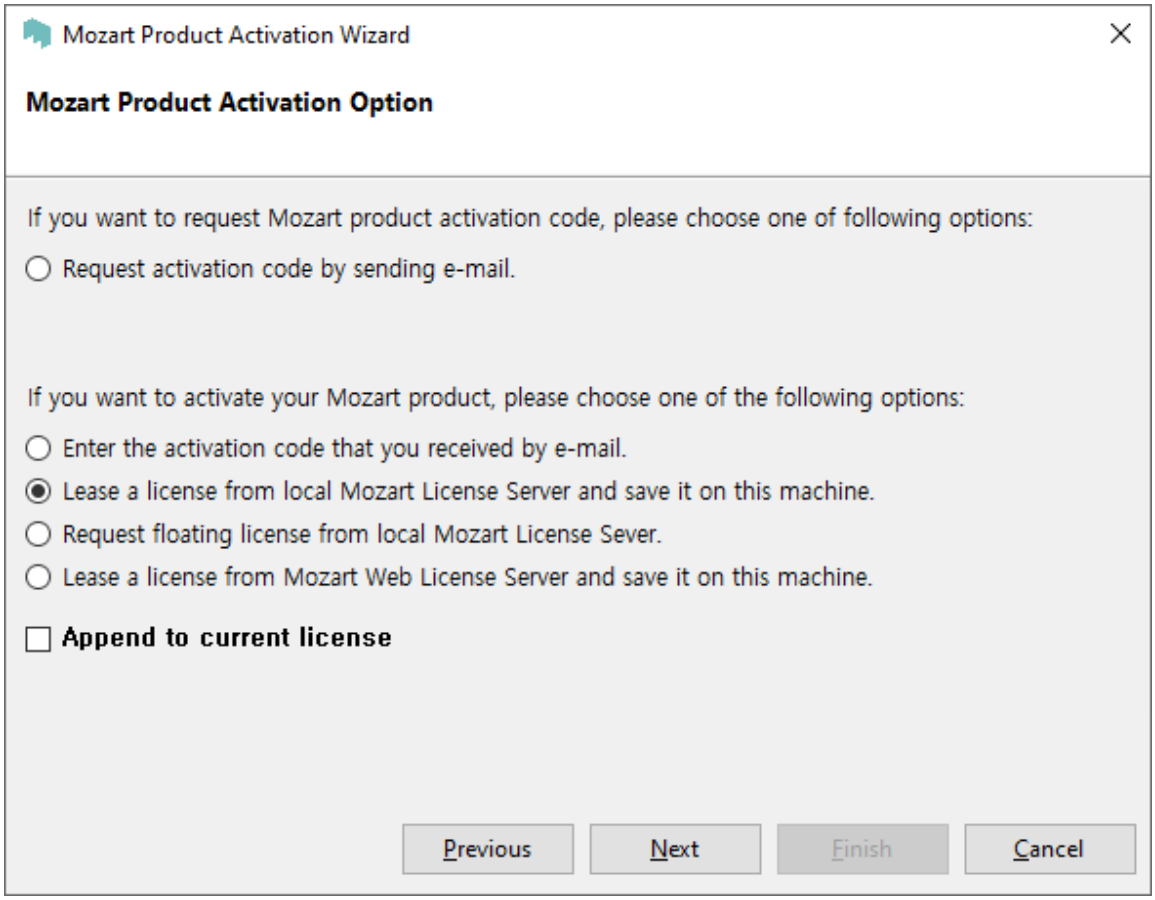
1. Mozart product activation option 선택 단계에서 "Enter the activation code that you received by e-mail" 을 선택한 후 **[Next]** 버튼을 클릭합니다.



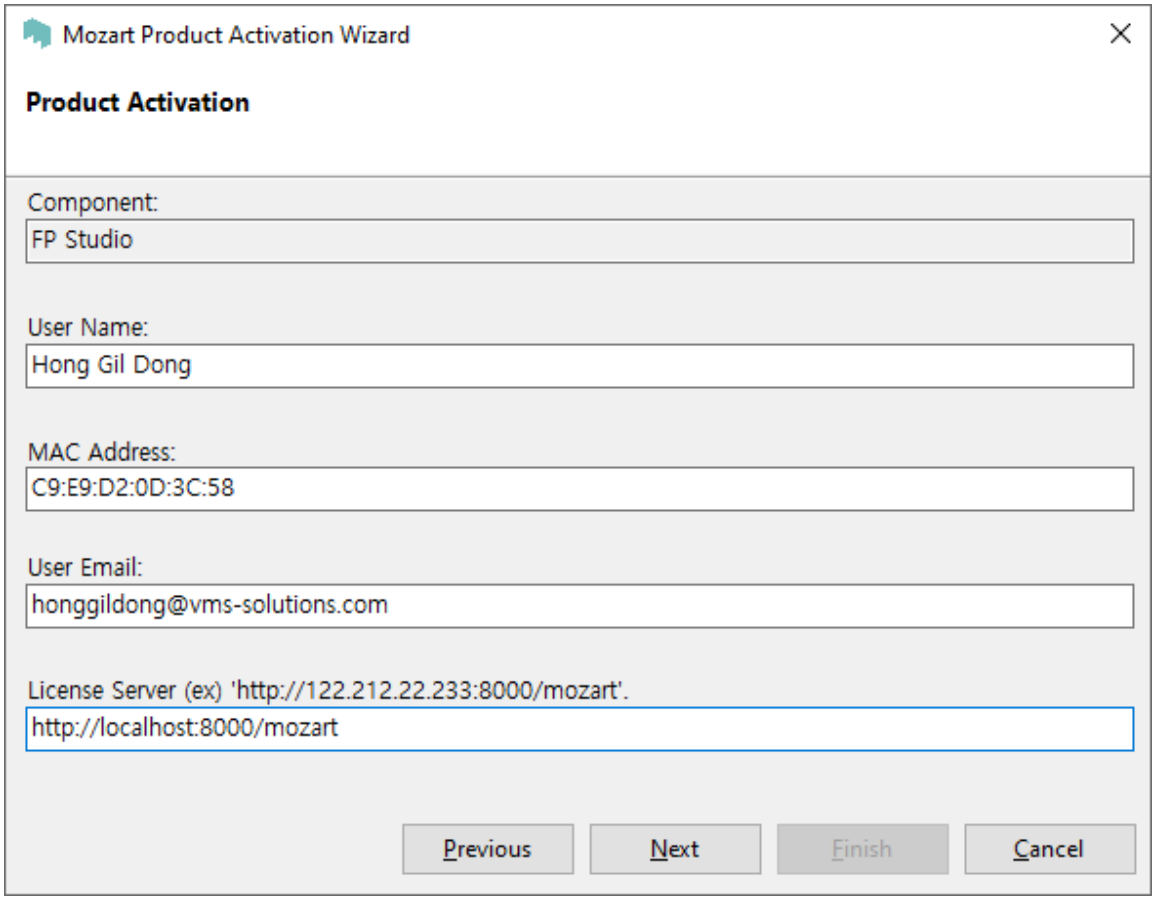
2. 수령한 Activation Code 를 복사하여 창에 붙여넣은 후 **[Next]** 버튼을 클릭합니다.
3. 정상적으로 활성화되면 활성화 정보가 표시됩니다.
4. **[Finish]** 버튼을 눌러 활성화를 마칩니다.

Local License Service 를 통한 Activation 방법

1. Mozart product activation option 선택 단계에서 하단의 “Lease a license from Mozart License Server” 옵션을 선택하고 **[Next]** 버튼을 클릭합니다.



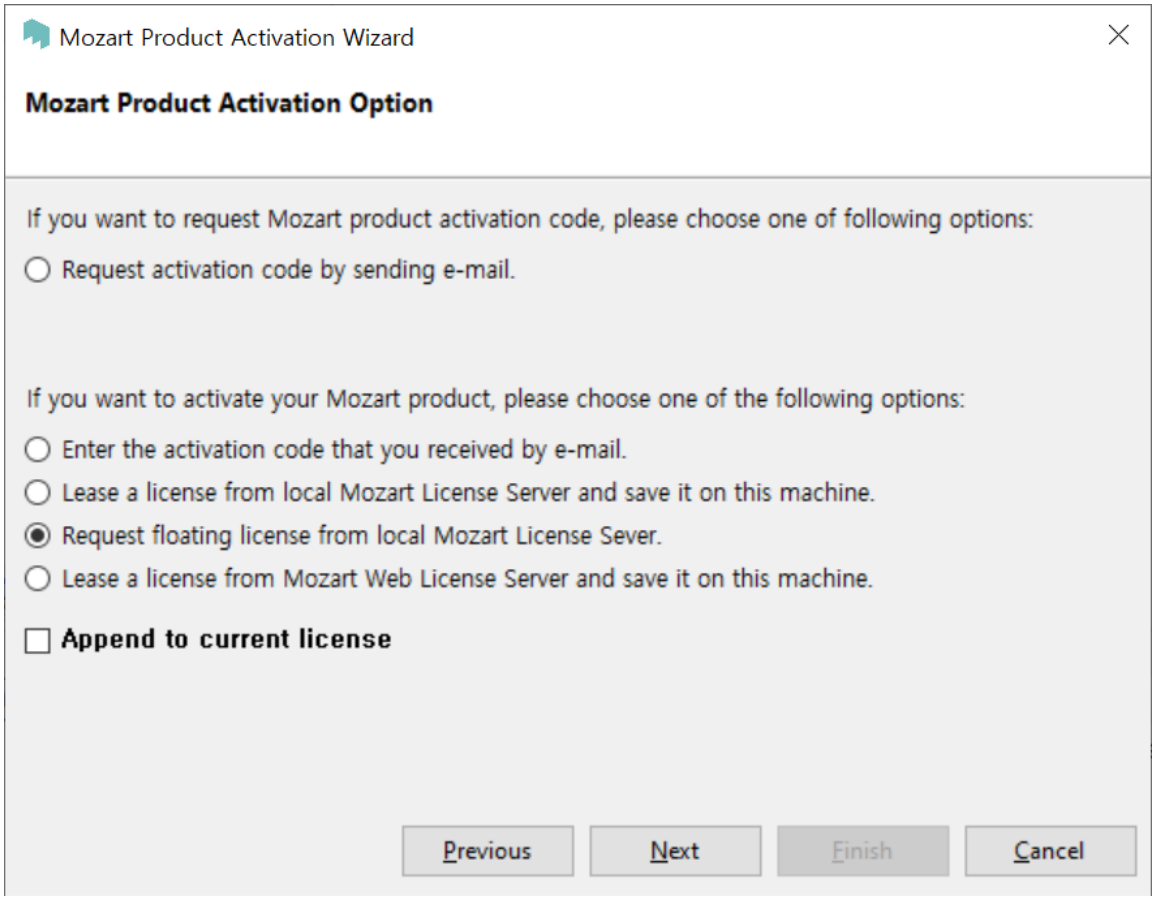
2. User Name, User Email 그리고 라이선스를 인증 받을 서버 주소를 입력합니다. 서버 주소 입력 시에는 Mozart Server 서비스의 포트 번호와 Application Name을 포함합니다. (예: http://1.1.1.1:8000/mozart)



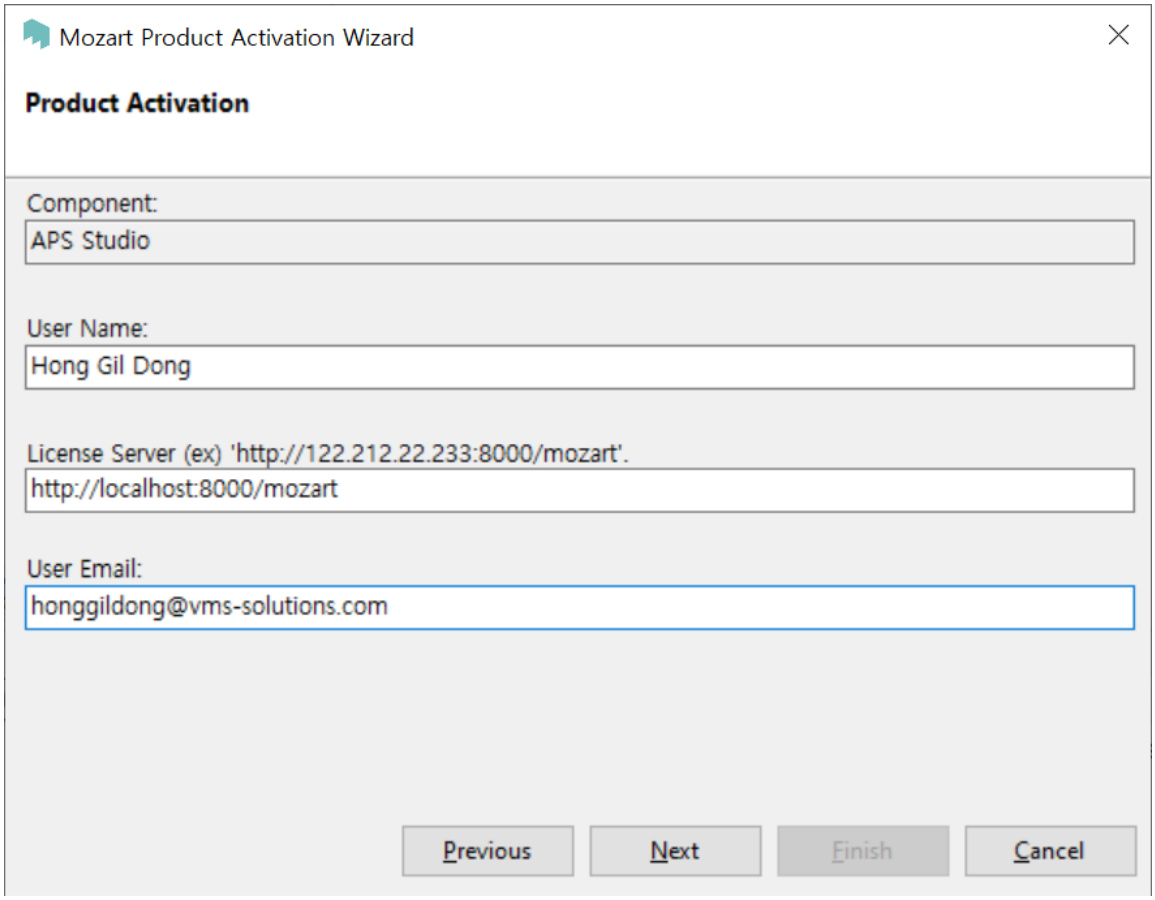
3. 정보를 다 입력하였다면 **[Next]** 버튼을 클릭합니다. 라이선스 인증이 완료되면 **[Finish]** 버튼을 클릭하여 **Mozart Product Activation Tool**를 종료합니다.

Floating License를 통한 Activation 방법

1. Mozart product activation option 선택 단계에서 하단의 “Request floating license from Mozart License Server” 옵션을 선택하고 **[Next]** 버튼을 클릭합니다.



2. User Name, User Email 그리고 라이선스를 인증 받을 서버 주소를 입력합니다. 서버 주소 입력 시에는 Mozart Server 서비스의 포트 번호와 Application Name을 포함합니다. (예: http://1.1.1.1:8000/mozart)



3. 정보를 다 입력하였다면 **[Next]** 버튼을 클릭합니다. 라이선스 인증이 완료되면 **[Finish]** 버튼을 클릭하여 **Mozart Product Activation Tool**를 종료합니다.

MOZART 라이선스 키 요청 방법

소개

MOZART 제품키 발급 절차가 변경됩니다. 2022.4.6 일부터는 MOZART 홈페이지에 가입된 사용자가 라이선스 발급기능을 통해 자동으로 라이선스를 받을 수 있습니다. 자세한 사용방법은 본 가이드를 참고해주시기 바랍니다.

기존과 달라진 점과 특징

라이선스 키 신청이 홈페이지로 이관되면서 기존 방식과 달라진 점은 다음과 같습니다.

- 키 자동 발급으로 인해 대기 시간 최대 5분으로 단축

- 라이선스 키 신청 현황을 실시간으로 확인 가능
- 요청서 작성 시 반복적인 정보 입력 최소화

사용자 구분

홈페이지에 가입된 회원에 한해서 라이선스 키 신청을 할 수 있습니다. 가입된 회원 중 일반 사용자와 사이트 관리자로 사용자 역할이 분류됩니다.

- **사이트 관리자**: 가입 회원 중 일부 임직원에게만 부여되는 역할로 본인 외 근무하는 사이트의 고객 또는 고객 자산의 서버의 라이선스 키를 대리 신청 할 수 있는 사용자입니다. 사이트 관리자는 본사 측에서 메일을 통해 대상자 지정을 요청합니다.
- **일반 사용자**: 사이트 관리자 외는 모두 일반 사용자입니다. 일반 사용자는 본인 외 같은 기업 소속 사용자의 라이선스 키를 대리 신청 할 수 있는 사용자입니다.

라이선스 키 자동 발급 여부 및 기간

라이선스 유형 그리고 제품 Component에 따라 라이선스 키 신청 시 자동 발급 여부와 기간이 결정됩니다.

				신청권한		
라이선스 유형	컴포넌트	사용기간(발급일로부터)	자동 발급 여부	사이트관리자	자사직원	고객사용자
구매 - 고객용	Mozart Studio	1년	Y	✓	✗	✓
	Mozart IDE	1년	구매수량 이내: Y	✓	✗	✓
	Mozart Server	영구				
	Mozart WISE	1년	N	✓	✗	✓
대여 - 임직원용	Mozart Studio	1년	Y	✓	✓	✗
	Mozart IDE					
	Mozart WISE					
	Mozart Server	1년	N	✓	✓	✗
평가	Mozart Studio	7일	Y	✓	✓	✓
	Mozart IDE					
	Mozart Server					
	Mozart WISE					

라이선스 유형

- **구매**: MOZART 제품을 구매하여 라이선스 계약이 체결된 **고객 전용** 라이선스 유형입니다. 홈페이지에 가입된 고객(일반 사용자) 및 사이트 관리자는 해당 유형으로 라이선스 키

신청이 가능합니다. 프로젝트 수행 용도로만 키 신청이 가능합니다.

- **대여**: **임직원 전용**이며 일반 사용자 및 사이트 관리자 모두 신청이 가능한 유형입니다. 프로젝트 수행 용도로만 키 신청이 가능하며 근무하는 사이트와 관계없이 임직원이 사용하는 경우 해당 유형으로 키 신청을 해야 합니다. 고객 자산의 PC에 라이선스 키를 신청하는 경우에는 사이트 관리자가 **Company**를 상주하는 고객사 변경 후 **[구매]** 유형에서 신청을 해야 합니다.
- **평가**: 제품 평가, 교육 수행 그리고 긴급 사용 용도를 위한 임시 라이선스 키를 발급 받을 수 있는 유형입니다. 홈페이지에 가입된 회원 **누구나** 신청이 가능합니다.

제품 라이선스 키 신청 방법

인터넷 익스플로러(IE)에서는 제품 라이선스 키 신청 페이지가 열리지 않습니다. 크롬이나 엣지 브라우저를 통해 접속하기 바랍니다.

1. 로그인 후 My Page 이동

라이선스 키 신청은 로그인 후 My Page에서 신청이 가능합니다. 사용자 정보 우측에 사용자가 속한 고객사와 라이선스 키를 발급 받을 수 있는 제품 목록으로 표시됩니다. 임직원 중 사이트 관리자는 기업을 변경하여 제품 키를 신청 할 수 있습니다.

> 💡 **도움말:**
> **[Company]** 목록에 고객 기업 또는 본인이 소속된 기업이 표시되지 않은 경우, 그리고 기업을 선택하였는데 제품이 표시되지 않은 경우 support@vms-solutions.com로 문의 주시기 바랍니다. 다음과 같은 사유로 표시되지 않을 수 있습니다.
>
> **[목록에 기업이 없는 경우]**
> 1. 사이트 관리자가 라이선스 키를 대리 신청 가능한 기업이 아님
> 2. 대상 기업에서 제품 구매를 하지 않음
>
> **[제품이 없는 경우]**
> 1. 기업은 등록되었으나 아직 제품 구매를 하지 않았거나 구매 정보가 등록되지 않음

2. 라이선스 키 요청서 작성

My Page에서 **[Get Key]**를 선택하면 대상 제품에 대한 라이선스 키 요청 페이지가 열립니다. 아래는 요청서 화면 구성입니다.

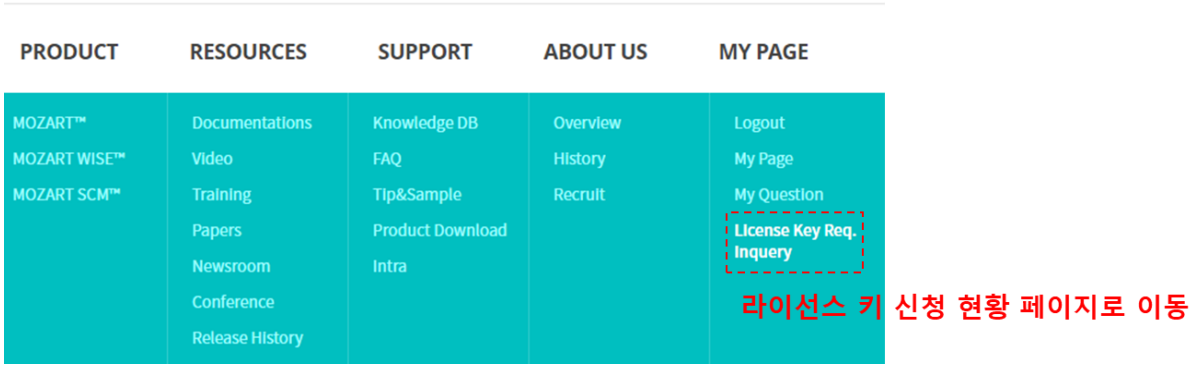
- **키 요청자 기본 정보**: 요청자가 선택한 제품과 라이선스 유형에 대한 정보를 표시합니다. 해당 영역은 수정이 불가하며 키를 요청하기 전 대상을 올바르게 선택하였는지 확인하기 위한 용도입니다.
- **키 요청서**: 키를 요청하기 위해 작성이 필요한 내용입니다.
- **라이선스 키 요청 목록**: 요청서 작성 후 **[추가]**를 선택하였을 때 목록에 추가되는 키 요청 내용입니다.
- **요청 삭제 버튼**: 요청서 추가 후 목록에 있는 요청서를 제거하기 위한 버튼입니다.
- **도움말**: MAC 주소 또는 컴퓨터 이름 확인 방법에 대해 설명하는 도움말과, 자동 발급 남은 키 수량에 대한 도움말을 표시합니다.

라이선스 키 요청은 다음과 같이 진행합니다.

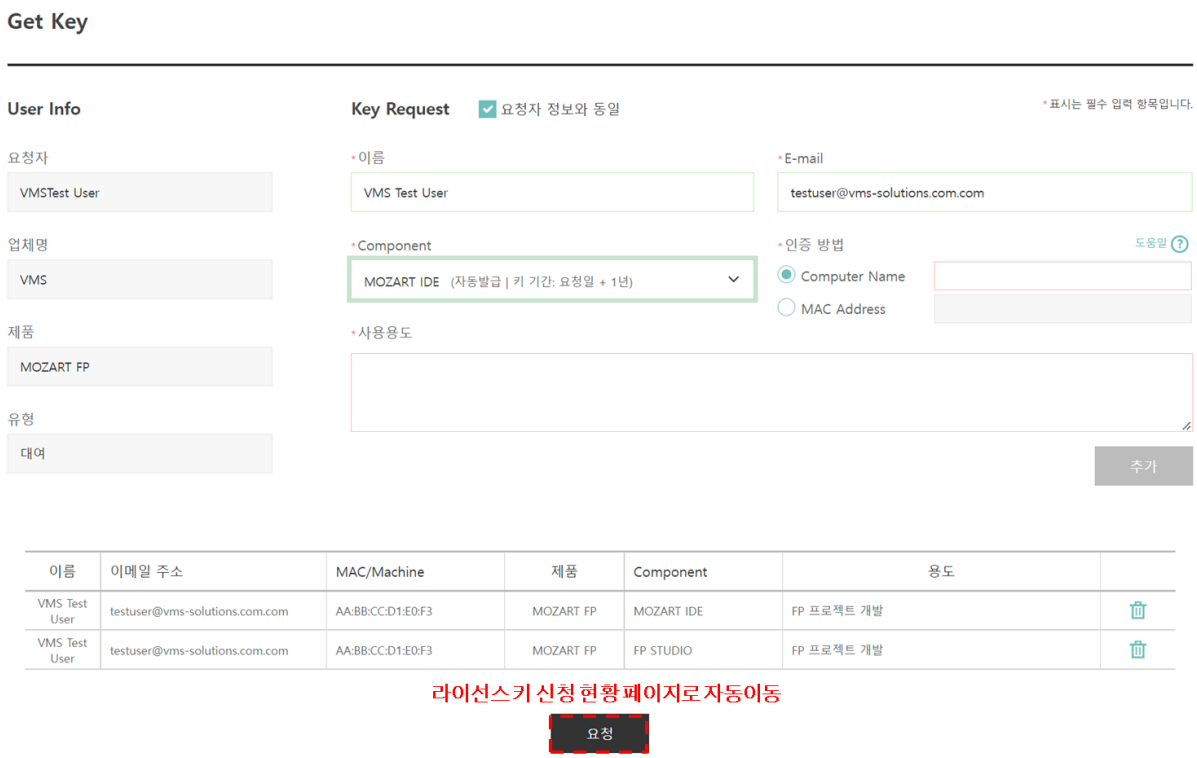
1. 키를 발급 받는 대상, 이메일 주소, 컴포넌트, 사용 용도, MAC 주소(**확인 방법**) 또는 컴퓨터 이름(확인 방법)에 대한 내용을 모두 작성합니다. 모든 항목에 대한 내용이 채워지면 **[추가]** 버튼이 활성화 됩니다.
2. 요청서 작성이 끝나면 **[추가]** 버튼을 눌러 요청 목록에 추가합니다. 다른 사용자 또는 컴포넌트에 대해 추가적으로 키 요청이 필요하면 1번 과정을 다시 진행하여 **[추가]** 버튼을 눌러 요청을 추가합니다.
3. 모든 요청이 추가 되었으면 **[요청]** 버튼을 눌러 라이선스 키를 신청합니다.

3. 라이선스 키 신청 현황

라이선스 키 신청 현황은 홈페이지 헤더 메뉴의 **My Page > License Key Req. Inquery**에서 확인 할 수 있습니다. 또한, 라이선스 키 요청서 페이지에서 키를 요청하게 되면 자동으로 라이선스 신청 현황 페이지로 이동합니다.



홈페이지 헤더 메뉴를 통해 이동하는 방법



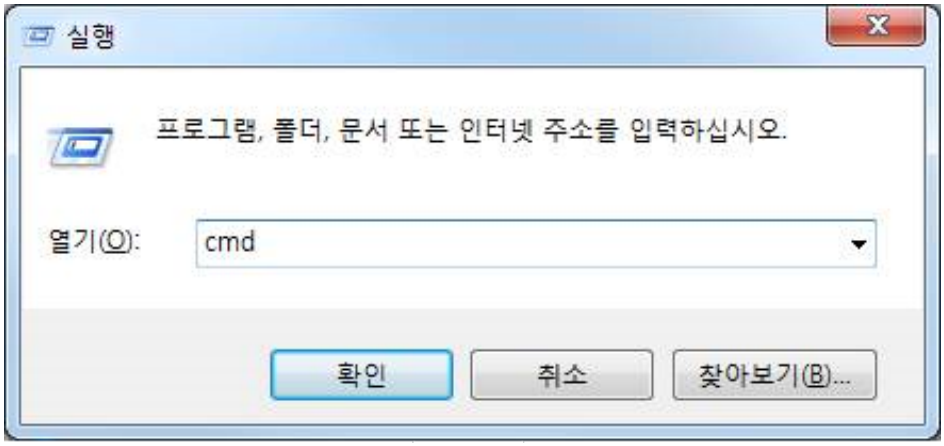
라이선스 키 신청 현황 페이지에서는 로그인한 사용자가 요청한 키의 전반적인 정보와 발급 상태를 조회 할 수 있습니다. 페이지 상단에서 발급 상태 그리고 신청 기간 조건을 설정하여 신청 현황을 조회 할 수 있습니다. 페이지를 처음 열 때는 최근 1주일에 대한 신청 현황을 조회 할 수 있습니다.

문의 및 문제 발생 시 연락

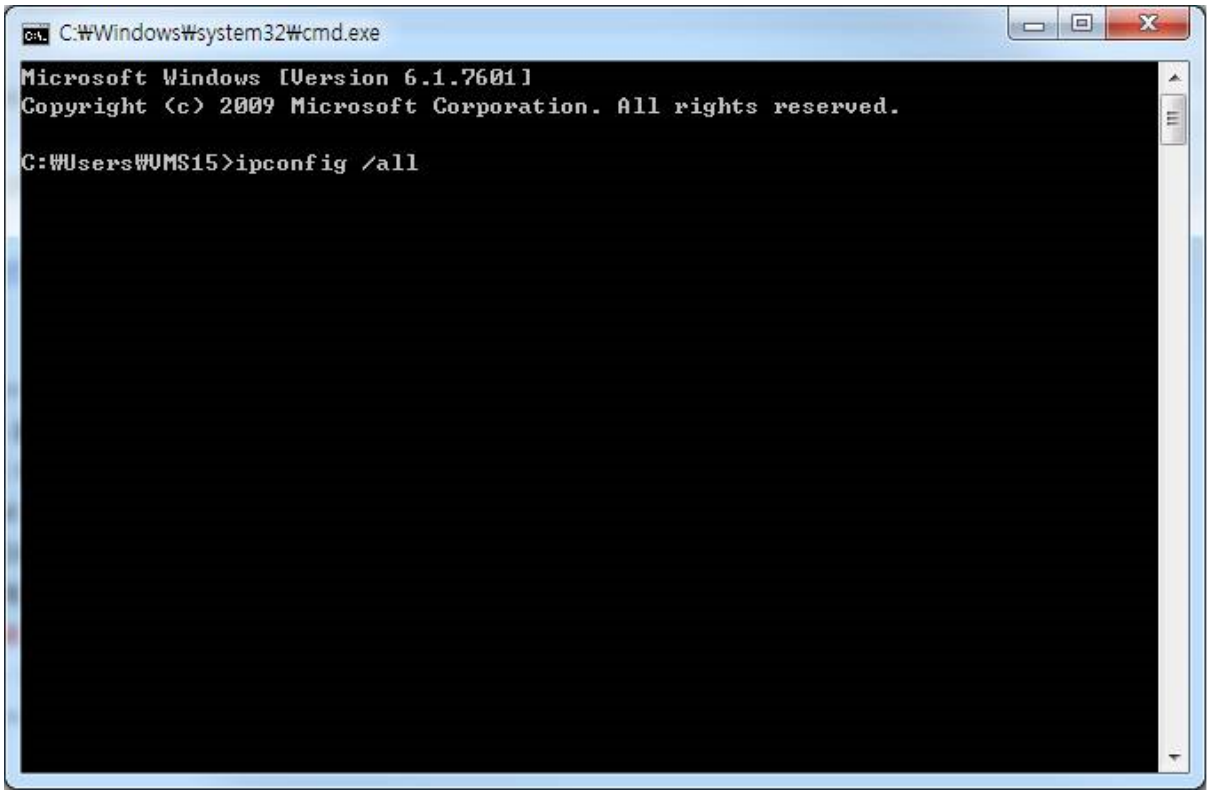
라이선스 키 신청에 대한 문의 사항 또는 문제 발생 시 support@vms-solutions.com으로 메일 보내주시기 바랍니다.

MAC 주소 확인 방법

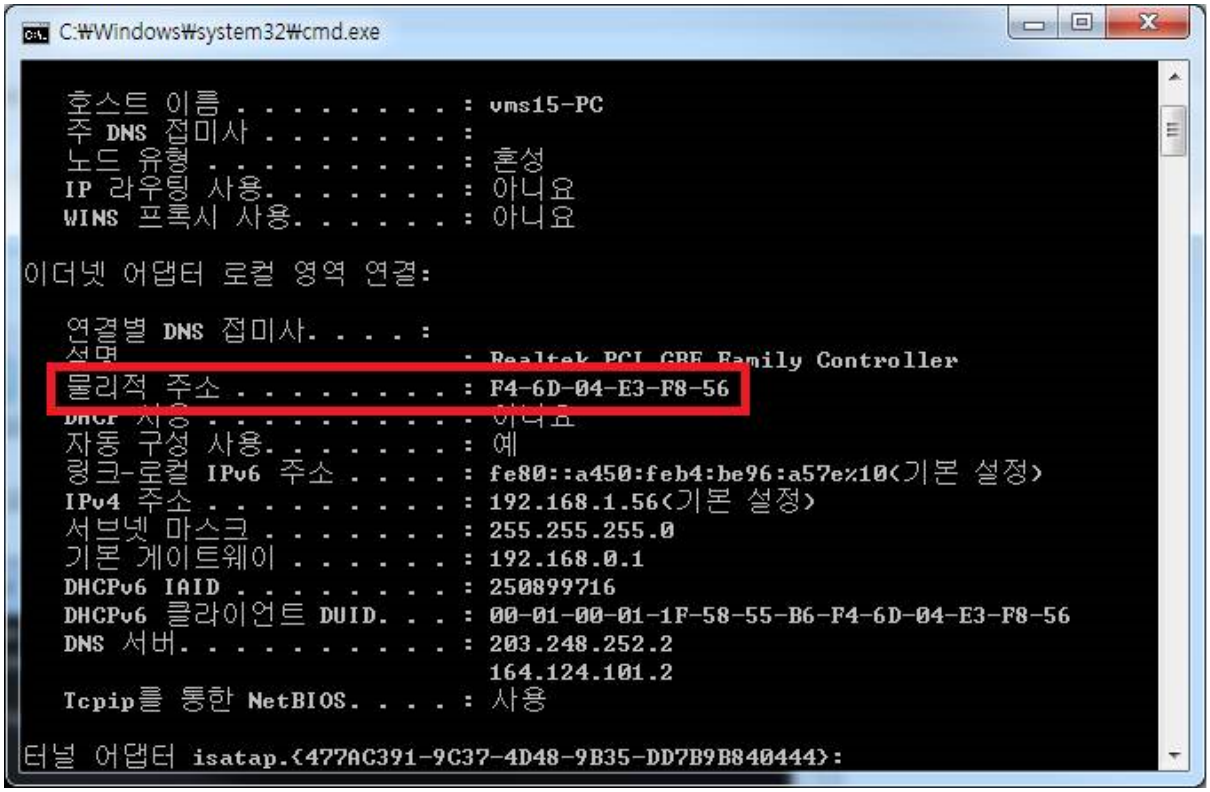
1. 키보드의 **윈도우 키 + r** 를 눌러 실행창에 “**cmd**" 입력 후 콘솔 창을 실행합니다.



2. 커맨드 창에 “**ipconfig /all**” 입력 후 Enter 키를 누릅니다.



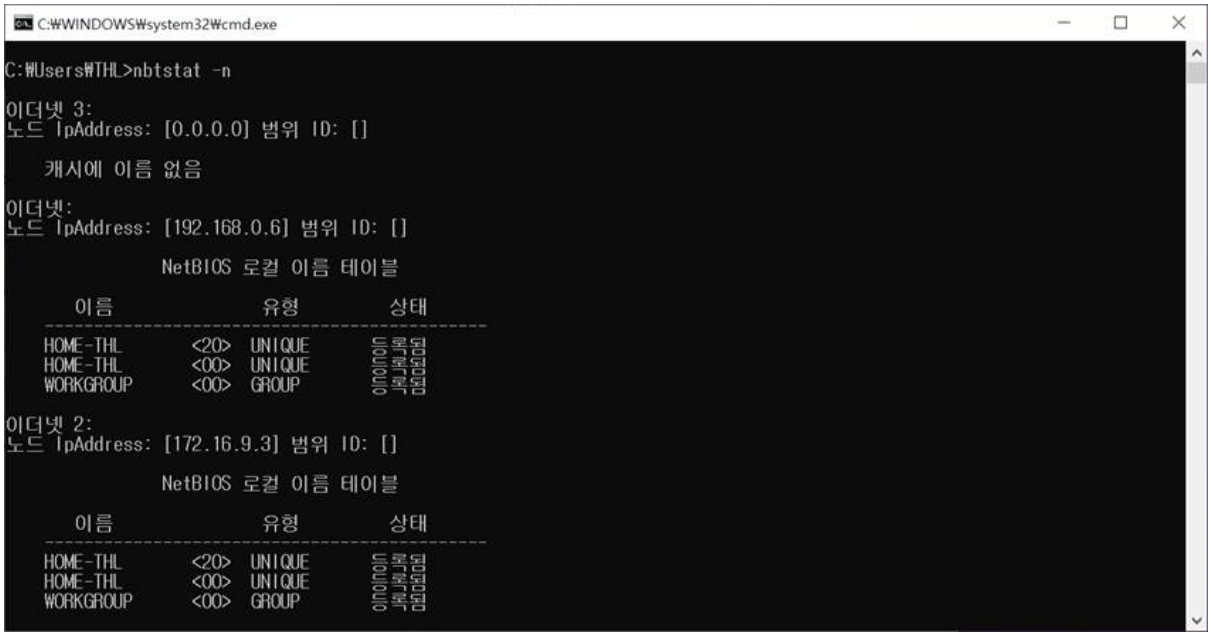
3. 명령을 실행 후 아래 그림과 같이 “물리적 주소”가 MAC 주소에 해당이 됩니다. LAN 카드가 두 개
있는 경우 네트워크에 물려 있는 LAN 카드의 MAC 주소를 보내주시면 됩니다.



컴퓨터 이름 확인 방법

라이선스 키 신청에 필요한 컴퓨터 이름은 아래와 같이 진행하면 확인이 가능합니다.

1. Windows + R 키를 눌러 실행창에 "**cmd"**라고 입력하여 명령 프롬프트를 실행합니다.
2. 명령창에 “**nbtstat -n**”을 입력 후 Enter 키를 누릅니다.
3. **[유형]**이 **UNIQUE**이고 **<00>** 접미사가 붙은 **[이름]**이 라이선스 인증 요청 시 전달되어야 하는 컴퓨터 이름입니다. (예: HOME-THL)



[그림 1. nbtstat 실행 결과]

컴퓨터 이름에 관하여

컴퓨터 이름은 윈도우 운영체제를 설치하는 과정에서 지정 할 수 있으며, 설치 후에도 이름을 변경 할 수 있습니다. 여기서 컴퓨터 이름은 UNIQUE 유형에 해당됩니다. 컴퓨터 이름은 최대 16바이트까지 지정할 수 있으며 15번째 바이트까지는 컴퓨터 이름에 해당되며 마지막 바이트는 제공되는 서비스의 접미사입니다. 즉, 컴퓨터 이름은 1자~15자까지만 작성할 수 있습니다. **컴퓨터 이름이 15자를 넘어가면 Mozart 제품 라이선스가 인증 되지 않으니 꼭 유의 하시기 바랍니다**.

[그림 1.]과 같이 <00>,<20>이 접미사이며 <00>은 컴퓨터 이름, <20>은 서버 서비스의 이름을 뜻합니다. 해당 PC에서 파일과 프린터 공유를 활성화 한 경우 <20> 접미사가 붙습니다.

컴퓨터 이름은 각 컴퓨터에 고유하게 사용되어야 하며 동일 네트워크상의 다른 컴퓨터의 이름과 중복되면 충돌이 발생합니다. 만약 충돌 발생 시에는 윈도우 제어판 또는 윈도우 설정을 통해 컴퓨터 이름을 변경해야 합니다.

참고자료:

컴퓨터, 도메인, 사이트 및 OUS 이름 지정 - Windows Server

이 문서에서는 Active Directory 디렉터리 서비스에 정의된 Windows, NetBIOS 도메인 이름, DNS 도메인 이름, Active Directory 사이트 및 US(조직 구성 단위)의 컴퓨터 계정에 대한 명명 규칙에 대해 설명합니

https://docs.microsoft.com/ko-kr/troubleshoot/windows-server/identity/naming-conventions-for-computer-domain-site-ou

제어판에서 컴퓨터 이름 변경 방법

1. **[제어판]** → **[시스템]**으로 이동합니다.
2. **[컴퓨터 이름, 도메인 및 작업그룹 설정]**에서 **[설정 변경]**을 선택합니다.
3. **[컴퓨터 이름]** 탭에서 **[변경]**을 선택하여 **[컴퓨터 이름/도메인 변경]** 팝업창에서 **[컴퓨터 이름:]**에 변경할 이름을 입력합니다.
4. **[OK]**를 눌러 변경사항을 저장하고, 컴퓨터를 재시작합니다.

윈도우 설정에서 컴퓨터 이름 변경 방법

이 방법은 윈도우 10, 윈도우 서버 2016 이상부터 사용 가능합니다.

1. Windows + X를 눌러 **[시스템(Y)]** 메뉴를 선택합니다.
2. **[디바이스 사양]** 항목에서 **[이 PC의 이름 바꾸기]**를 선택합니다.
3. 변경할 컴퓨터 이름을 입력하고 **[다음]**을 선택합니다.
4. 컴퓨터를 재시작합니다.

GETTING STARTED

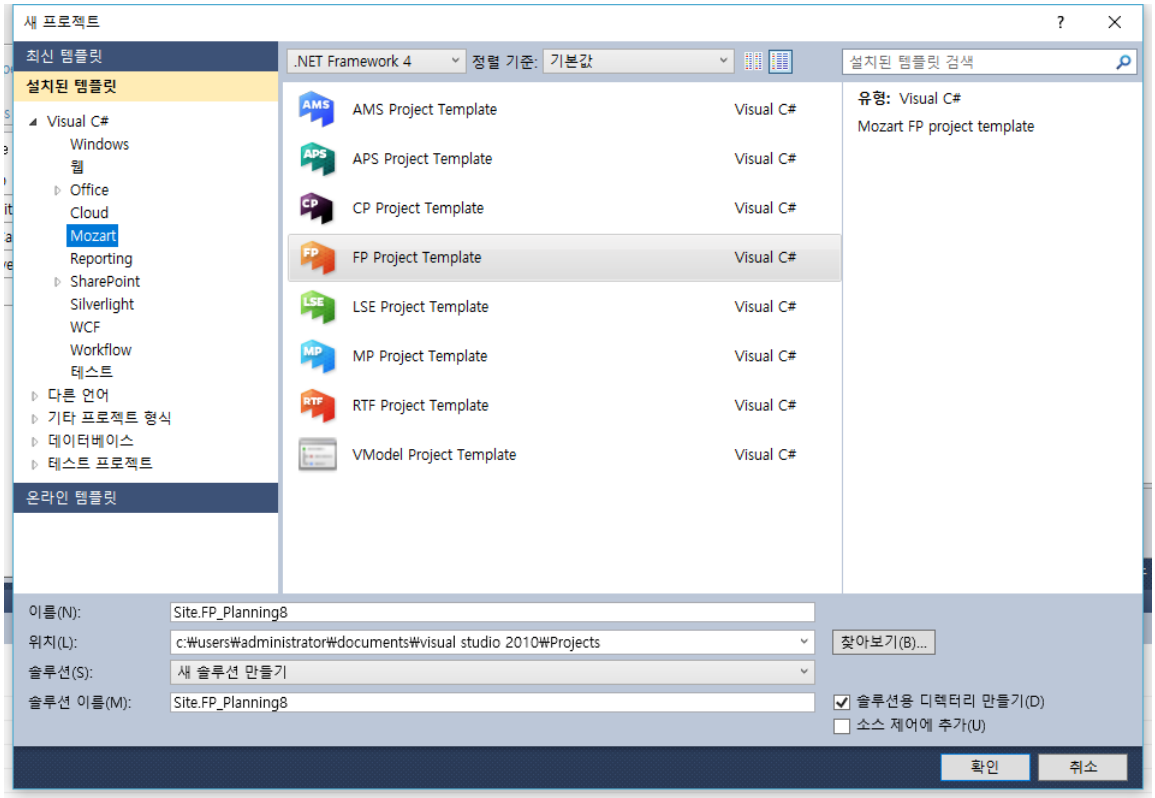
Hello World Example

본 예제는 MOZART IDE 를 통해 MOZART Project 를 생성하고 MOZART 개발 Logic 의 실행환경인 MOZART Studio 를 통해 실행 Parameters 를 받아 이를 Studio 의 Console 창에

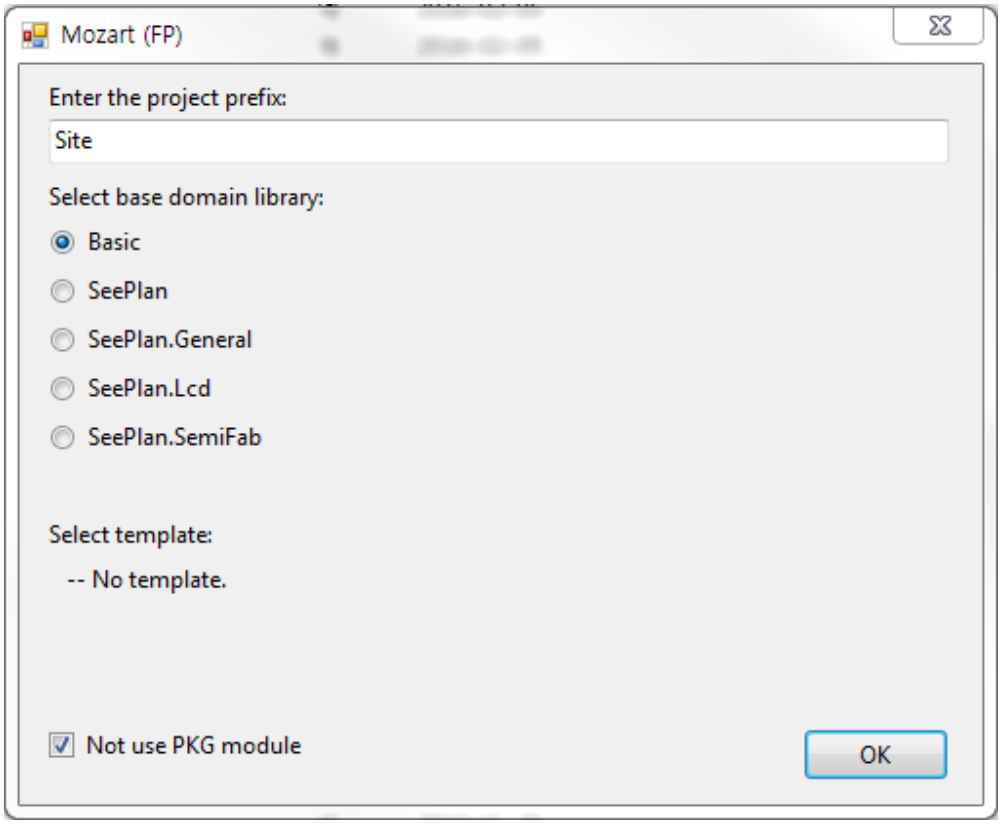
Log 형태로 출력하는 과정을 수행함으로써 MOZART Project 구성 및 실행환경을 이해하도록 구성되어 있습니다.

Project 생성하기

1. Visual Studio의 **[파일 > 새로만들기 > 프로젝트]** 메뉴를 선택합니다.
2. [새 프로젝트] 다이얼로그에서 Visual C# 의 Mozart Template 을 선택하고 솔루션명, 생성경로를 입력하고 **[확인]** 버튼을 클릭합니다.



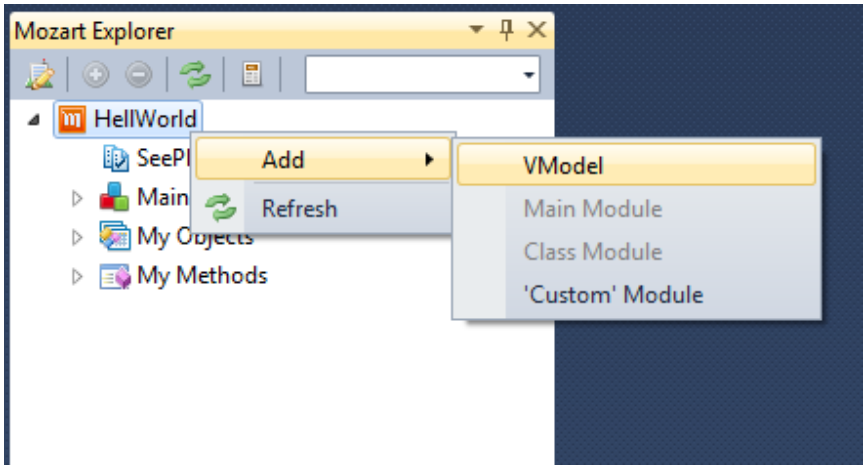
3. Project 에서 사용할 Domain Library 를 선택합니다. 본 예제에서는 Library 를 사용하지 않으므로 **[OK]** 버튼을 눌러 다음을 진행합니다.



입력한 Project 명으로 MOZART Explorer 에 MOZART Project 가 생성된 모습을 확인하고, Visual Studio 의 Solution Explorer 에 Project 가 생성된 모습을 확인합니다.

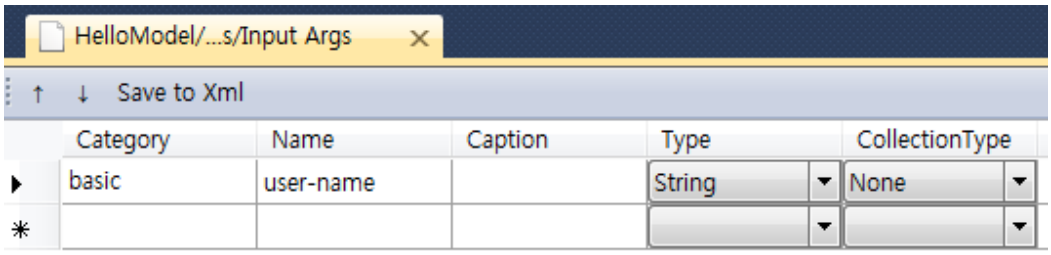
Argument 설정하기

1. MOZART Explorer 의 최상위 노드인 Project 를 선택하고 오른쪽 마우스 버튼을 통해 **[Add > Vmodel]** 메뉴를 실행합니다.
2. Model 명을 “HelloModel”로 입력하고 **[OK]** 버튼을 클릭하면, MOZART Explorer 에 입력한 명칭으로 VModel 이 생성됩니다.



3. 생성된 Model 을 열어 **[Arguments > Input Args]** 를 더블 클릭하여 Argument 입력창을 활성화합니다.

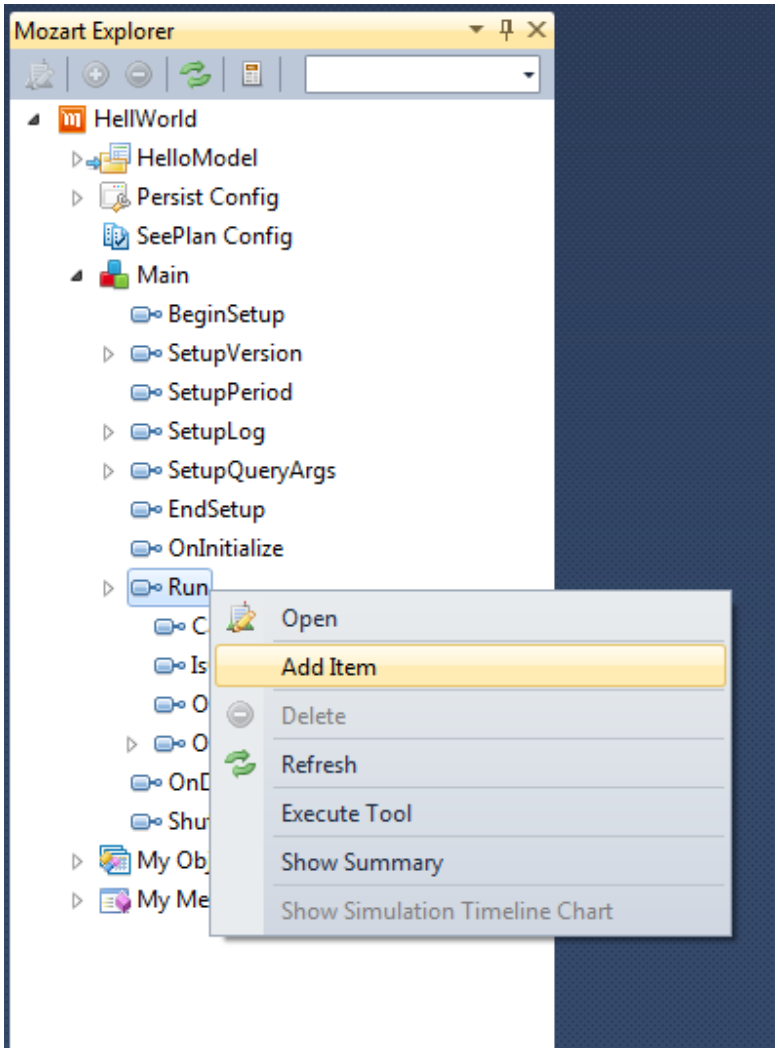
4. Argument 의 Name 을 “user-name” 으로 입력하고 Type 을 String 으로 선택합니다. 이는 추후 실행시점에 본인 이름을 입력할 용도로 입력변수를 생성하는 작업입니다.



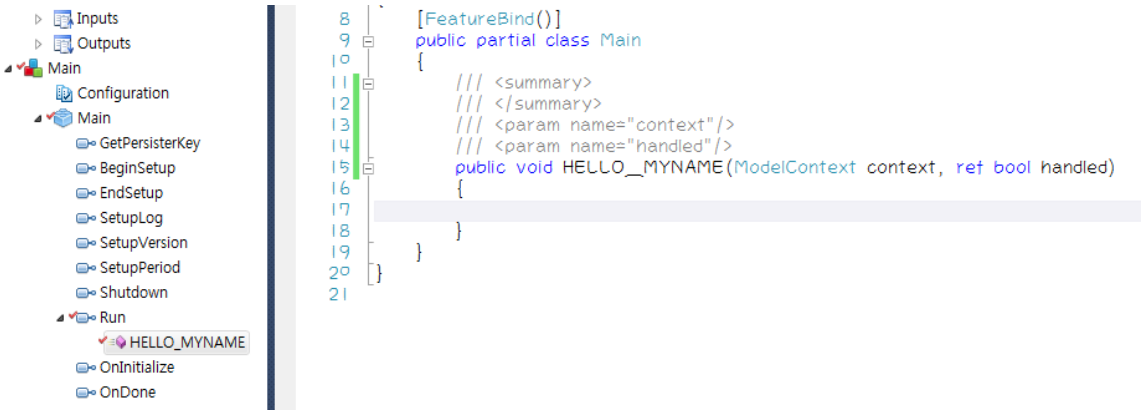
Log 출력을 위한 Main 함수 작성하기

1. MOZART Explorer 의 Main Module 을 열고 Main Component 를 확장한 후 Run 함수를 재정의합니다.

2. Run 함수 노드에서 **[Add Item]** 팝업메뉴를 실행합니다.



3. Add FE Method 창에서 Method 명을 “HELLO_MYNAME” 으로 입력하면 함수가 생성됩니다.



4. 생성된 함수 원형 아래 Log 출력을 위한 코드를 작성합니다.

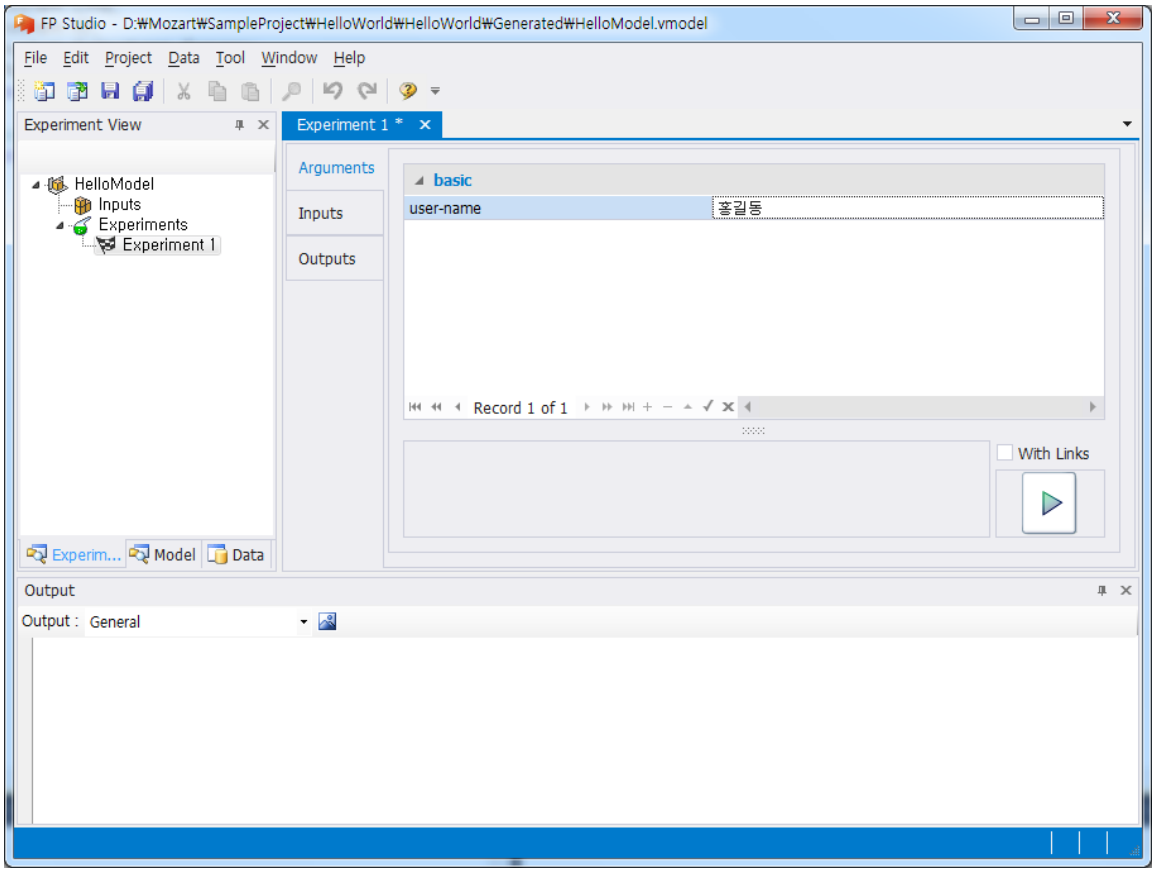
Log 출력 함수 작성하기

```
public void HELLO_MYNAME(ModelContext context, ref bool handled)
{
    string myname = InputMart.Instance.GlobalParameters.user_name ;
    string msg = "Hello " + myname ;
    Logger.MonitorInfo(msg);
}
```

5. 입력한 코드를 모두 저장합니다.

Model 실행하기

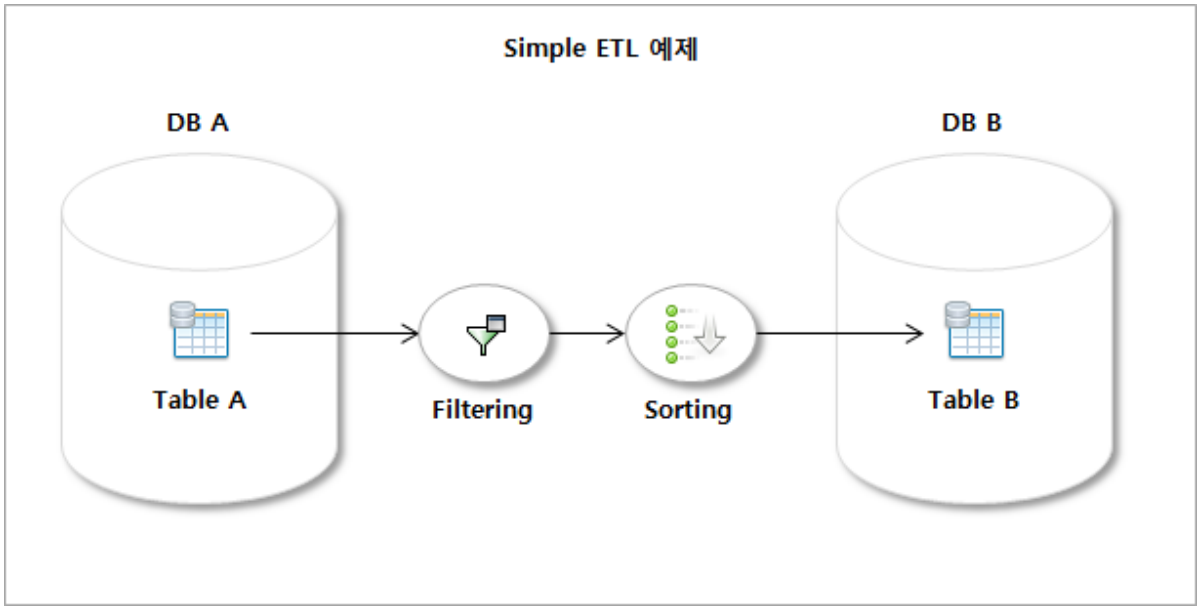
1. 메뉴의 [디버그 > 디버깅 시작] 을 선택하거나 **“F5”** 를 눌러 디버깅을 시작합니다.
2. 디버깅 시작 시 MOZART Studio 가 실행됩니다.
3. MOZART Studio 가 실행되면, 왼쪽 Experiment View 에서 **[Experiments > Experiment 1]** 노드를 더블 클릭하여 Argument 입력창을 띄웁니다.
4. Input Args 로 정의한 user-name 에 값을 입력한 후, 하단의 실행 버튼을 클릭합니다.



5. 하단의 Output 창에서 실행 Log 중 Main 함수에 구현한 Log 가 출력되는 것을 확인합니다.

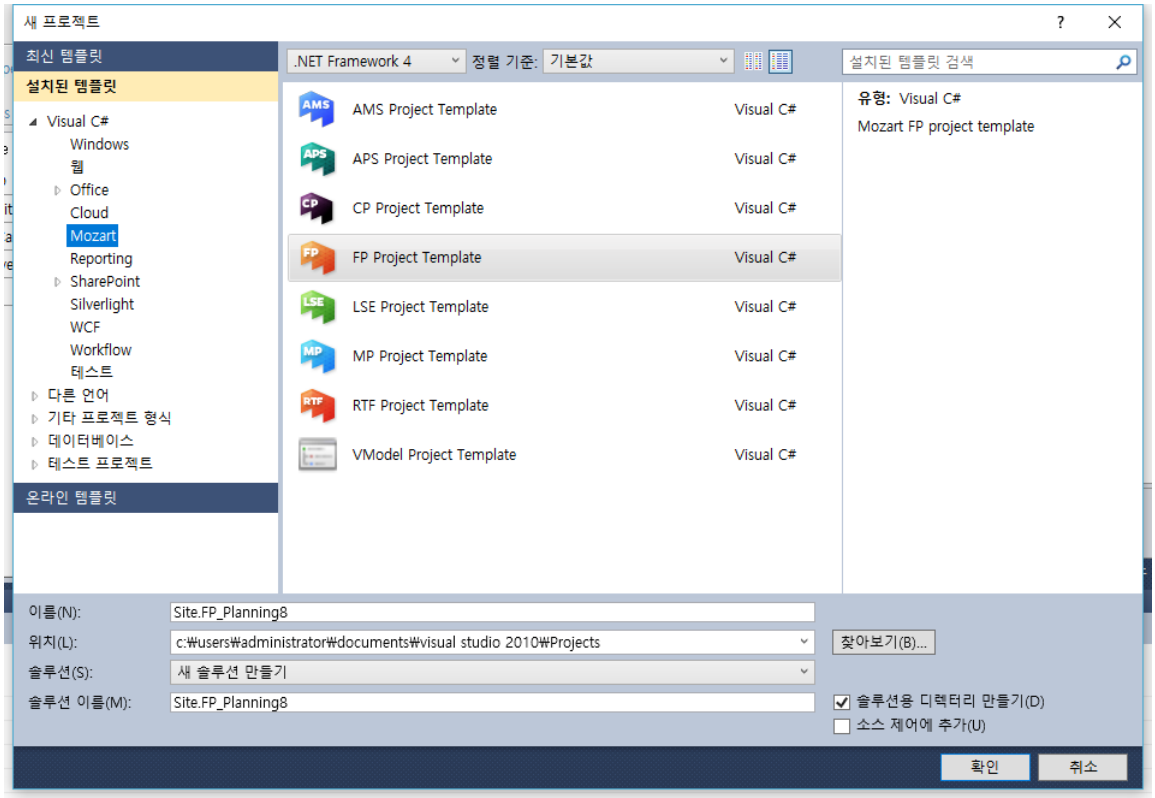
Simple ETL Example

본 예제는 MOZART IDE 를 통해 MOZART Project 를 생성하고 두 개의 서로 다른 DB간에 데이터를 추출(Extract), 변환(Transform), 저장(Load)하는 작업을 MOZART Model 의 Database, Input, Output 및 Persist 기능을 사용하여 구현하는 방식에 대해 설명합니다. 또한, 개발된 Dll 을 Server 에 Job 으로 등록하고 실행 Schedule 을 등록하여 실행하는 과정을 통해 MOZART 운영환경을 경험할 수 있도록 구성되었습니다.

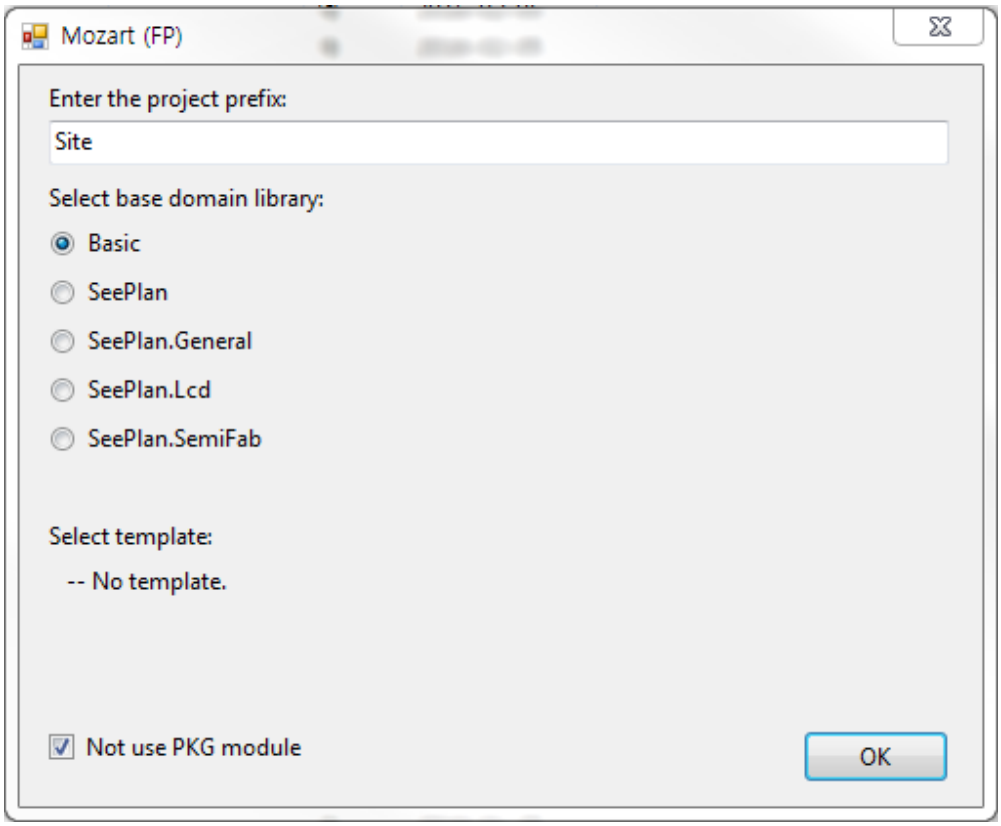


Project 생성하기

1. Visual Studio의 **[파일 > 새로만들기 > 프로젝트]** 메뉴를 선택합니다.
2. [새 프로젝트] 다이얼로그에서 Visual C# 의 Mozart Template 을 선택하고 솔루션명, 생성경로를 입력하고 **[확인]** 버튼을 클릭합니다.



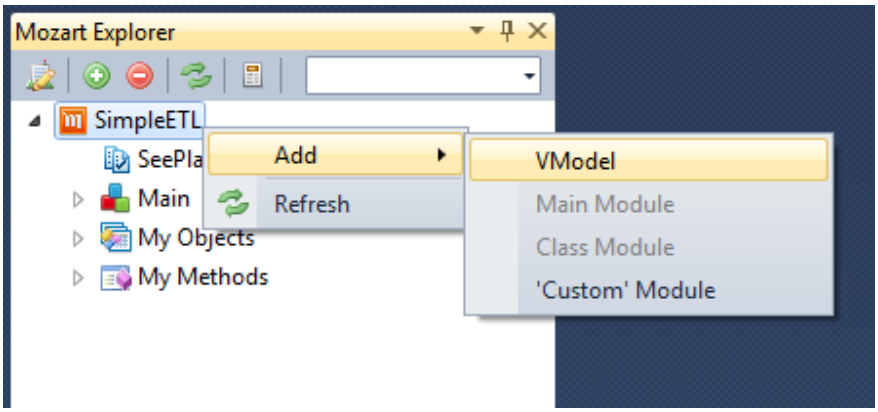
3. Project 에서 사용할 Domain Library 를 선택합니다. 본 예제에서는 Library 를 사용하지 않으므로 **[OK]** 버튼을 눌러 다음으로 진행합니다.



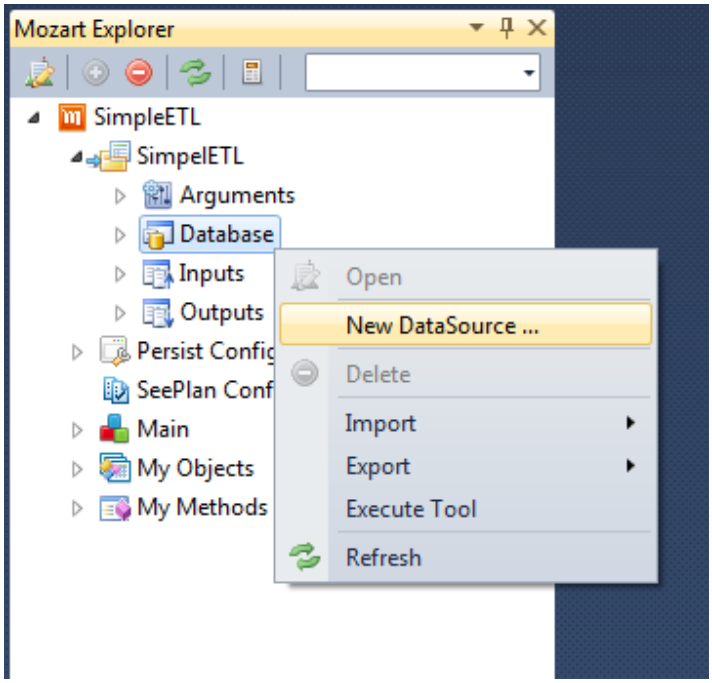
4. 입력한 Project 명으로 MOZART Explorer 에 MOZART Project 가 생성된 모습을 확인하고, Visual Studio 의 Solution Explorer 에 Project 가 생성된 모습을 확인합니다.

Database 등록하기

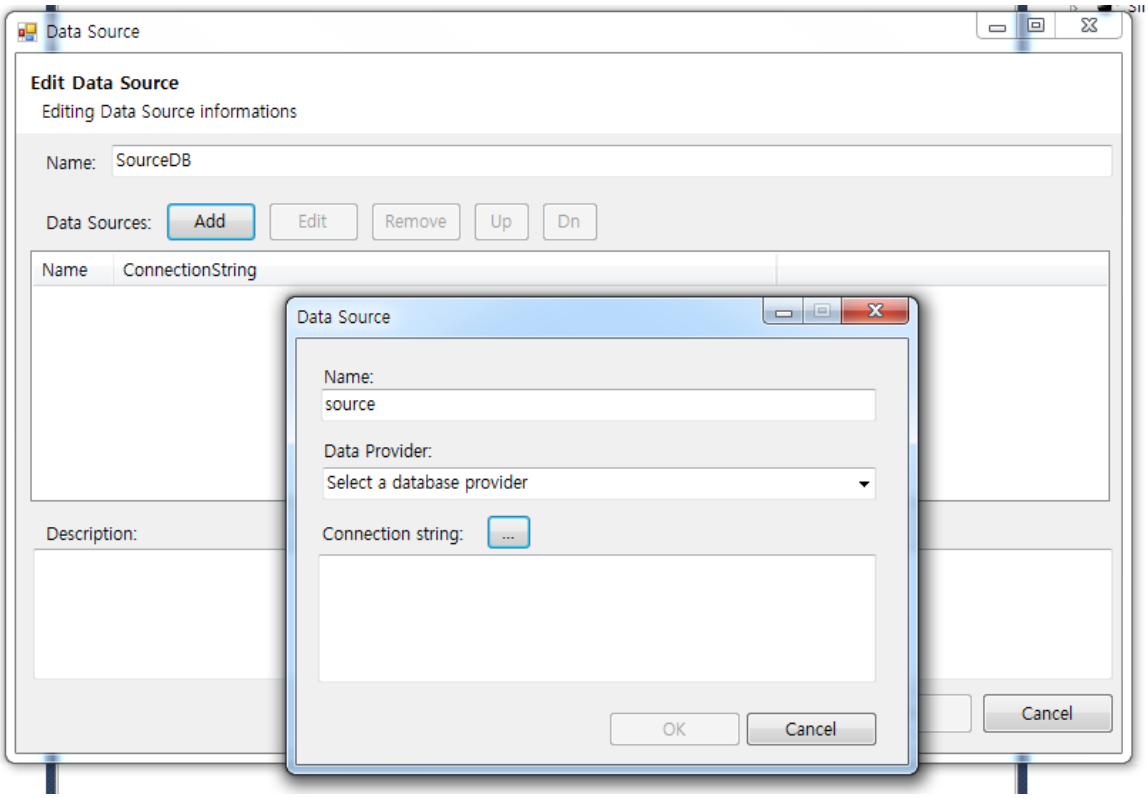
1. Database 등록을 위해 먼저 SimpleETL Logic 개발을 위한 Model 을 생성합니다.
2. MOZART Explorer 의 최상위 노드인 Project 를 선택하고 오른쪽 마우스 버튼을 통해 **[Add > Vmodel]** 메뉴를 실행합니다.



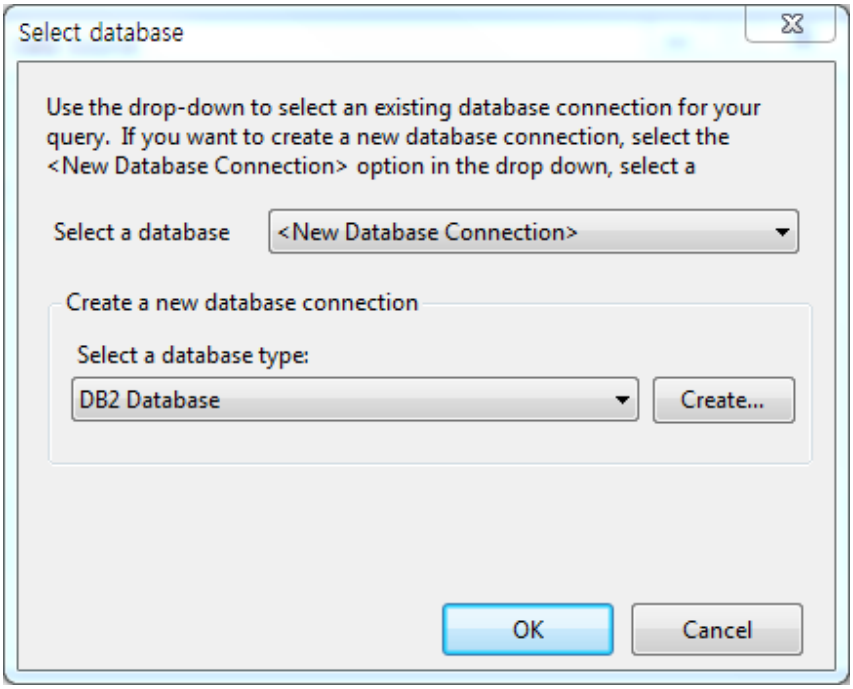
3. Model 명을 “SimpleETL” 로 입력하고 [OK] 버튼을 클릭하면 입력한 명칭으로 MOZART Explorer 에 VModel 이 생성됩니다.
4. Model 의 하위 노드인 Database 노드에서 **[New DataSource…]** 팝업메뉴를 실행하여 Database 를 추가합니다. Source 가 되는 DB A 와 Target 이 되는 DB B 두 개의 Data Source 를 추가합니다.



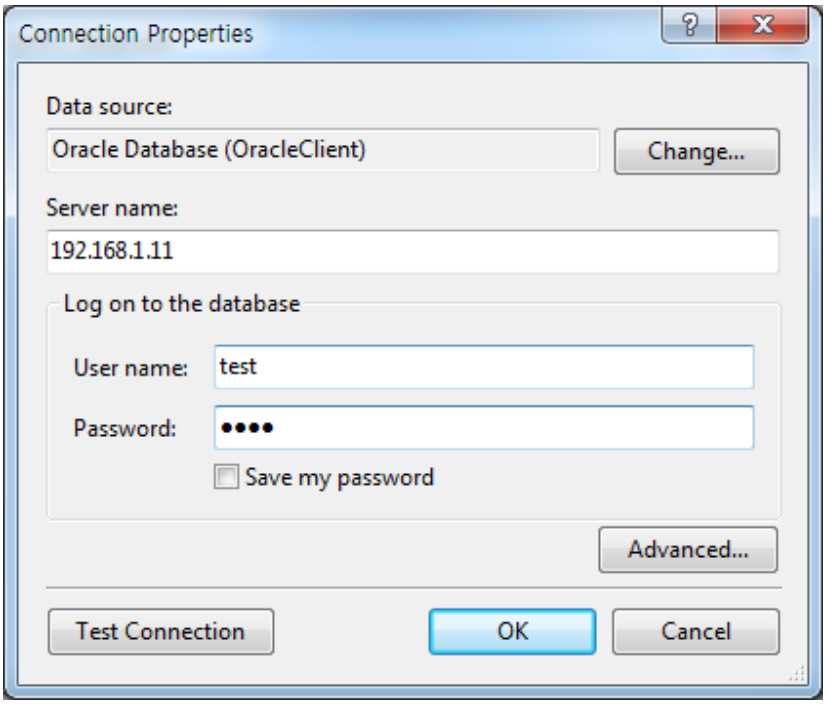
5. Data Source 추가 창에서 연결할 Database 명을 입력하고 **[Add]** 버튼을 사용하여 Data 연결정보를 입력합니다.



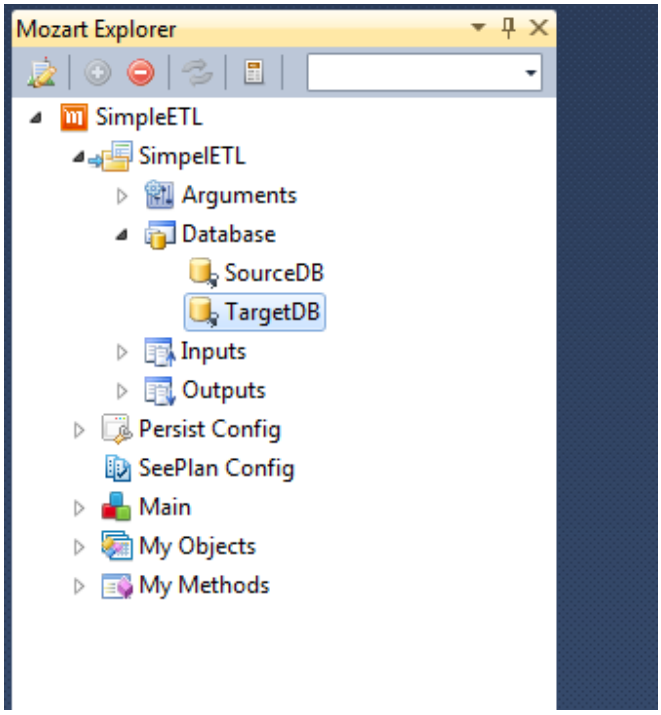
6. **Connection string** 을 입력하거나 **[…]** 버튼을 이용하여 연결정보를 입력합니다.
7. 버튼을 이용하는 경우, 먼저 database type 을 선택하고 **[Create…]** 버튼을 눌러 접속정보를 입력합니다.



8. Server 명칭 혹은 IP 주소와 User, Password 정보를 입력합니다.

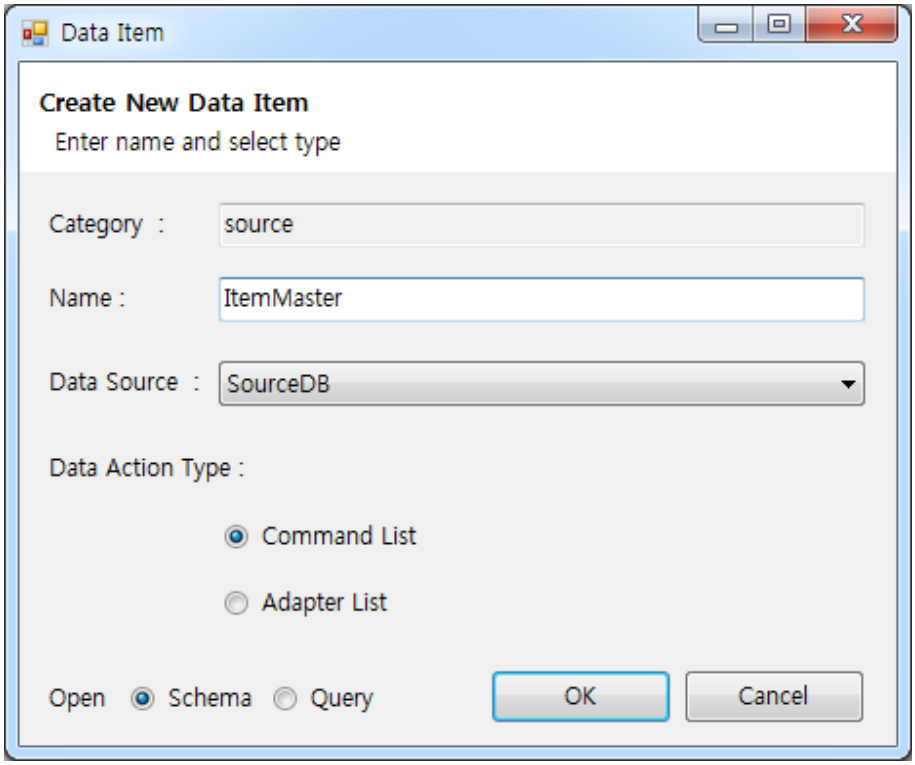


9. Source DB (DB A) 를 추가한 방식과 동일하게 Target DB (DB B) 에 대한 Database 를 등록합니다.



Input, Output Schema 정의하기

1. Model 의 하위 노드인 Inputs 을 선택하고 오른쪽 마우스 버튼을 통해 **[New DataGroup…]** 팝업메뉴를 실행하여 Data Group 명이 “source” 인 Data Group 을 생성합니다.
2. Outputs 노드에도 동일한 방식으로 Data Group 명이 “target” 인 Data Group 을 생성합니다.
3. 생성한 ‘source’ Data Group 에서 오른쪽 마우스 버튼을 통해 **[New DataItem…]** 팝업메뉴를 실행하여 Table 명이 “ItemMaster” 인 Source Table **(Table A)** 을 생성합니다.



예제 Data Table 및 Data 생성하기

상기 작업을 하기 위해 먼저 등록한 각 Source, Target Database 에 Table 을 생성해 두어야 합니다.

본 예제에서는 임의로 대상이 되는 양쪽 DB 에 아래와 같은 Format 의 동일한 Data 를 임의로 작성해두고 사용합니다. 이 부분은 필요에 의해 다시 정의해서 사용하실 수 있습니다.

SOURCE DB

```
CREATE TABLE TBL_ITEM_MASTER
(
  ITEM_GROUP VARCHAR2(10) ,
  ITEM_ID   VARCHAR2(10),
  ITEM_NAME VARCHAR2(30),
  ITEM_PRICE NUMBER
)
```

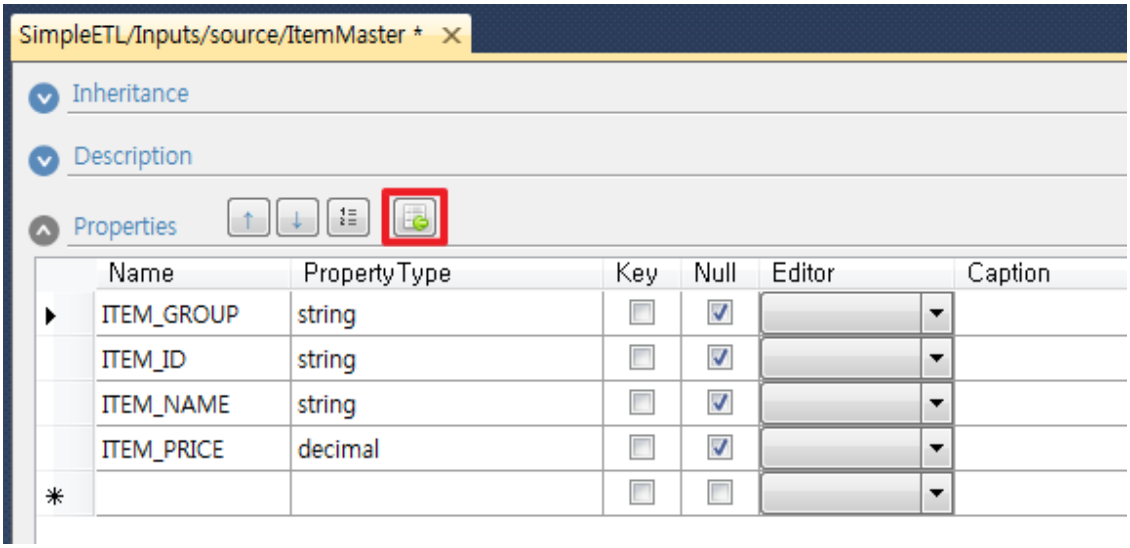
TARGET DB

```
CREATE TABLE TBL_ITEM_SELECTED
(
  ITEM_GROUP VARCHAR2(10) ,
  ITEM_ID   VARCHAR2(10),
  ITEM_NAME VARCHAR2(30),
```

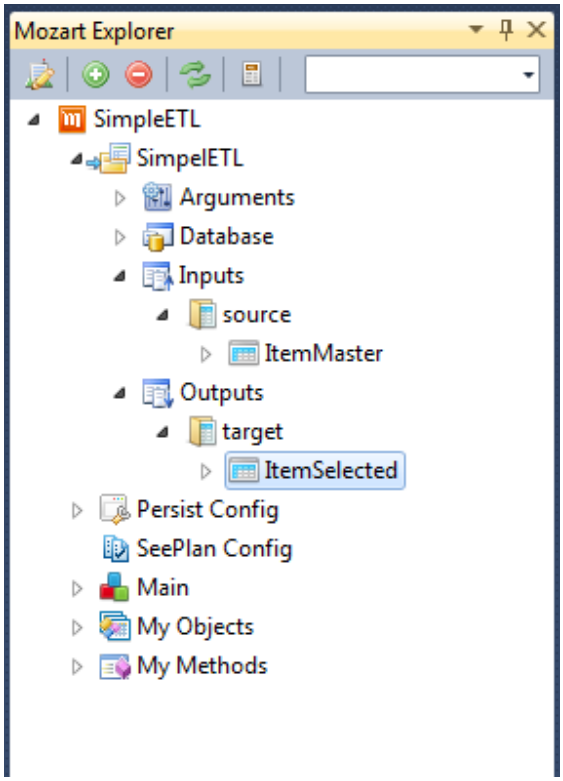
```
  ITEM_PRICE NUMBER
)
```

Table 을 생성한 후, Source DB 에 적절한 Data 를 입력하여 준비합니다. Target DB 에는 Source DB 에서 가공된 Data 가 저장되므로 Data 를 입력하지 않아도 됩니다.

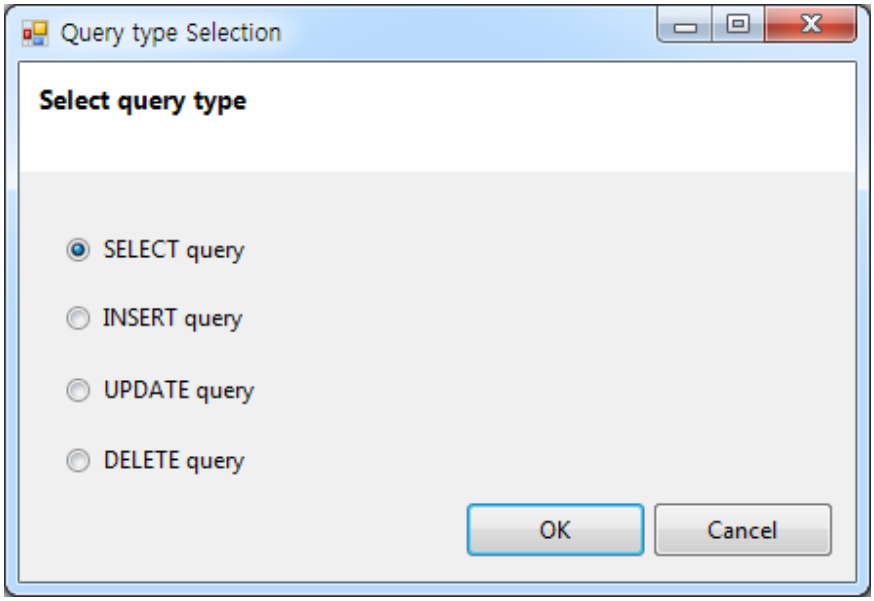
1. 생성된 DataItem 을 더블 클릭하면 Schema 를 편집할 수 있는 화면이 활성화됩니다.
2. 이때, 상단의 빨간 상자에 표시된 아이콘(Schema Extractor)을 클릭하여 생성해둔 Table 의 Schema 를 Import 합니다. 사전에 등록한 Data Source 를 선택하고 Table 을 선택한 후, 가져오려는 Columns 를 선택합니다.



3. 본 예제에서는 'ItemMaster' Table 에 있는 모든 Columns 이 필요하므로 **[Select All]** 버튼을 클릭한 후, **[Finish]** 버튼을 클릭하면 선택한 Table 과 동일한 Schema 가 생성됩니다.
4. 동일한 방식으로 'target' Data Group 에도 Table 명이 "ItemSelected" 인 Target Table **(Table B)** 을 생성합니다. 이 때, Data Source 설정을 Target DB (DB B)로 해야한다는 것에 유의합니다.

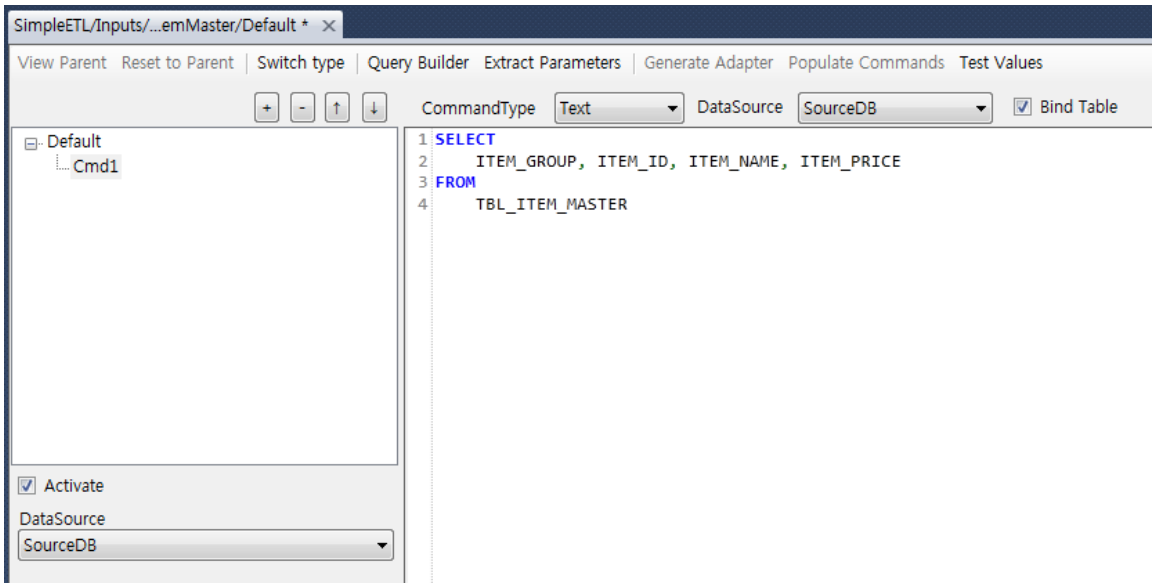


5. Source Data Table 추출을 위한 Query 를 작성합니다. 먼저, **[ItemMaster > Default]** 를 더블 클릭하여 DataAction 편집창을 엽니다.
6. 편집창 좌측 하단의 Activate 체크 여부와 Data Source 설정을 확인 한 후, Query 를 작성합니다.
7. Query 를 작성하는 방법에는 두 가지가 있습니다. 직접 Query 를 작성하거나 **[Query Builder]** 버튼을 이용하여 Query 를 작성합니다.
8. 버튼을 이용하는 경우, Query Type Selection 화면에서 작성하고자 하는 Query Type 을 선택합니다.

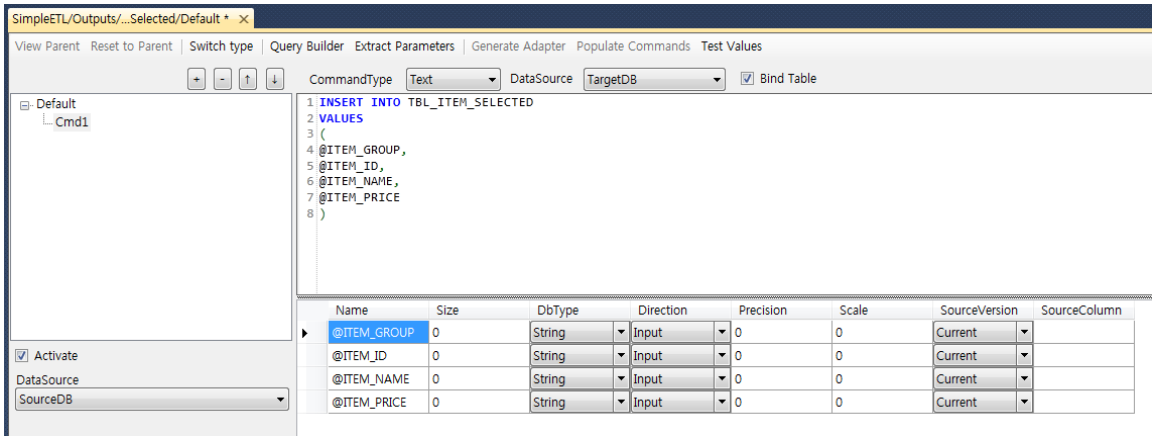


9. Query Type 을 선택하고 **[OK]** 버튼을 클릭한 후, Table 의 Schema 를 Import 할 때와 같은 화면이 나타나면, Table 을 선택한 후 가져오려는 Columns 를 선택합니다.

10. 본 예제에서는 'ItemMaster' Table 에 있는 모든 Columns 이 필요하므로 **[Select All]** 버튼을 클릭한 후, **[Next] - [Finish]** 버튼을 클릭하여 Query 작성을 완료합니다.



11. Target Table 인 "ItemSelected" DataItem 에 대해서는 Data Source 를 Target DB 로 설정한 후, Insert Query 를 작성합니다. 입력이 되는 변수 앞에 "@" 접두어를 붙여 Query 를 작성한 후 상단의 **[Extract Parameters]** 를 실행하면 하단에 입력한 Parameters 가 추출됩니다.



12. 모두 저장을 하면, Logic 개발을 위한 Inputs, Outputs 의 정의가 끝납니다.

변환 및 Data 저장 Logic 개발

1. Model 의 Inputs 노드 하위에 Persist Config 를 더블 클릭하여 Input Persist 편집창을 엽니다. Input Persist 에 대한 사용법은 [Input Data Persist Configuration]을 참조하시기 바랍니다.

2. Input 의 Tree 를 펼쳐 소스가 되는 DataItem 을 선택하고 오른쪽 기능 중 “On after load item” 에 체크한 후 //아이콘 버튼을 클릭하여 New Method 입력창에 처리 함수명을 입력합니다.

3. 생성된 함수를 편집하기 위해 //아이콘 버튼을 사용하여 편집기를 열고 변환 코드를 입력합니다.

4. 변환코드는 두 가지로, 먼저 ITEM_PRICE 가 500,000 미만인 아이템은 제외하도록 하는 코드를 작성합니다.

   Filtering Logic 구현

```
public bool OnAfterLoad_SourceItem(ItemMaster entity)
{
    if (entity.ITEM_PRICE < 500000)
        return false;
    return true;
}
```

> ⚠️ **Warning**
> ItemMaster (혹은 ItemSelected) 에 대한 NameSpace 오류 발생 시, ‘using SimpleETL.Inputs(Outputs);’ 를 추가하거나 ‘SimpleETL.Inputs.ItemMaster(ItemSelected)’ 를 직접 입력하여 해결할 수 있습니다

5. 그리고 모든 Data 를 Filtering 한 후 호출되는 Logic 을 구현하기 위해 “Executing action” 에 체크한 후 동일한 방식으로 Handler 함수를 생성하고 ITEM_PRICE 에 대해 내림차순 Sorting 을 합니다.

   Sorting Logic 구현

```
 public void OnAction_SourceItem(Mozart.Task.Execution.Persists.IPersistContext cont
ext)
 {
     InputMart.Instance.ItemMaster.DefaultView.Sort = "ITEM_PRICE DESC";
 }
```

6. 마지막으로 로딩된 Data 를 Output (ItemSelected) 에 옮기는 작업을 수행하도록 하는 코드를 작성합니다.

   Output 저장코드 생성

```
 public void OnAction_SourceItem(Mozart.Task.Execution.Persists.IPersistContext cont
ext)
 {
     InputMart.Instance.ItemMaster.DefaultView.Sort = "ITEM_PRICE DESC";
     foreach(ItemMaster it in InputMart.Instance.ItemMaster.DefaultView)
     {
         ItemSelected add = new ItemSelected() ;
         add.ITEM_GROUP = it.ITEM_GROUP;
         add.ITEM_ID = it.ITEM_ID;
         add.ITEM_NAME = it.ITEM_NAME;
         add.ITEM_PRICE = it.ITEM_PRICE;
         OutputMart.Instance.ItemSelected.Add(add);
     }
 }
```

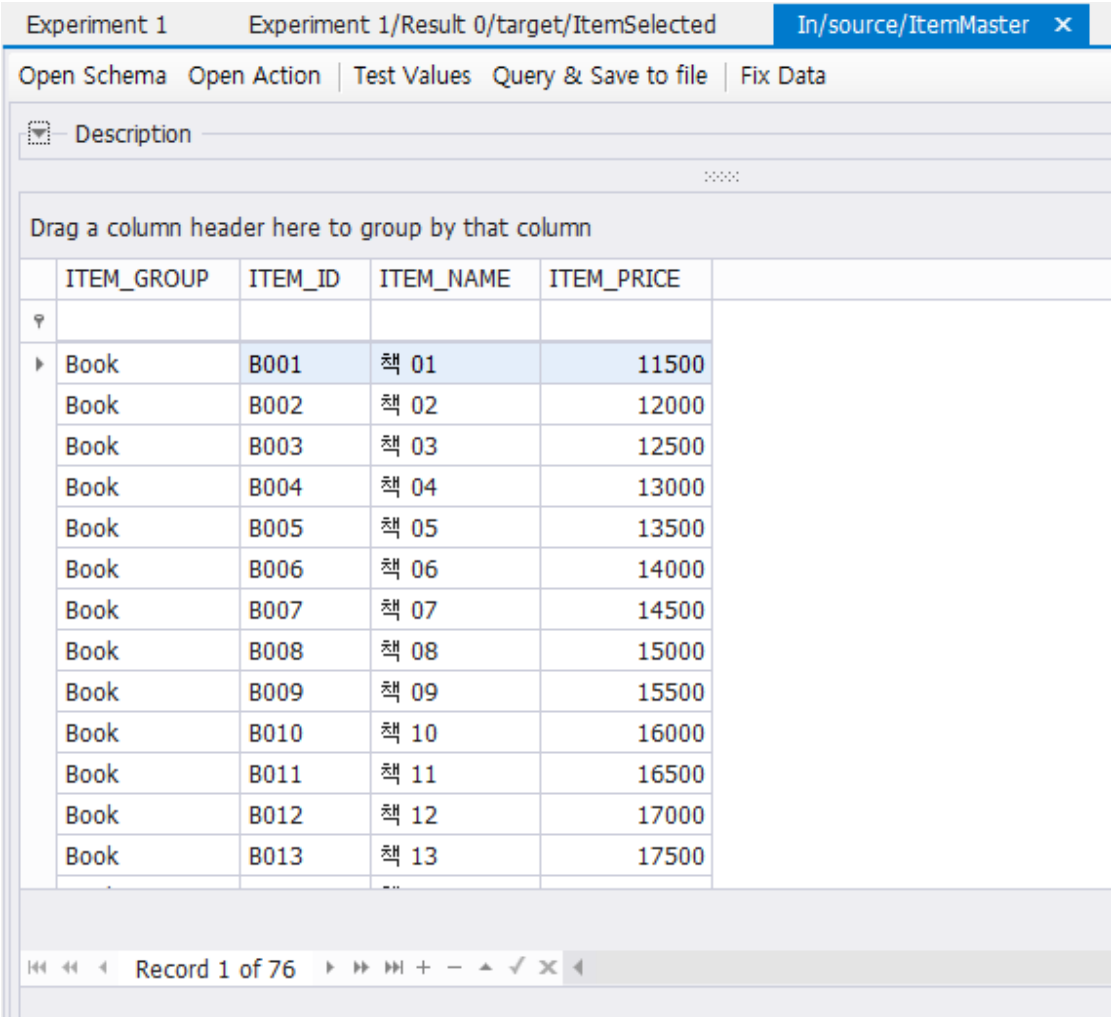
7. 처리가 정상적으로 되는지 확인하기 위해 Model 을 실행해봅니다.

Model 실행

1. 메뉴의 **[디버그 > 디버깅 시작]**을 선택하거나 **“F5”**를 눌러 디버깅을 시작합니다.
2. 디버깅 시작 시 MOZART Studio 가 실행됩니다.

3. MOZART Studio 가 실행되면 왼쪽의 Experiment View에서 **[Input > source > ItemMaster]** 노드를 더블 클릭하여 Input Data 를 입력할 수 있는 창을 엽니다.

4. Input Data 를 확인하기 위해 입력창 상단 부근의 **[Query & Save to file]** 메뉴를 실행합니다.



ITEM_GROUP	ITEM_ID	ITEM_NAME	ITEM_PRICE
Book	B001	책 01	11500
Book	B002	책 02	12000
Book	B003	책 03	12500
Book	B004	책 04	13000
Book	B005	책 05	13500
Book	B006	책 06	14000
Book	B007	책 07	14500
Book	B008	책 08	15000
Book	B009	책 09	15500
Book	B010	책 10	16000
Book	B011	책 11	16500
Book	B012	책 12	17000
Book	B013	책 13	17500

5. Data 가 정상적으로 로드 되었으면, **[Experiments > Experiment 1]** 을 더블 클릭해서 실행창을 띄운 후 실행 버튼을 누릅니다.

6. 실행 Log 가 작성되고 완료되면 Experiment 1 하위에 Result 0 폴더가 생깁니다. Result 0 폴더를 선택하면 Model 의 Outputs 에서 정의한 Output 폴더 (target) 와 그 하위에 Output 을 저장하기 위한 DataItem (ItemSelected) 에 가공 결과가 저장된 것을 확인할 수 있습니다. Input 의 경우 총 76건의 Data 가 Sorting 되어 있지 않은 채로 있었으나, Output 은 총 26건의 Data 로 아래 그림과 같이 내림차순 Sorting 된 결과를 보여줍니다.

Experiment 1 | Experiment 1/Result 0/target/ItemSelected | In/source/ItemMaster

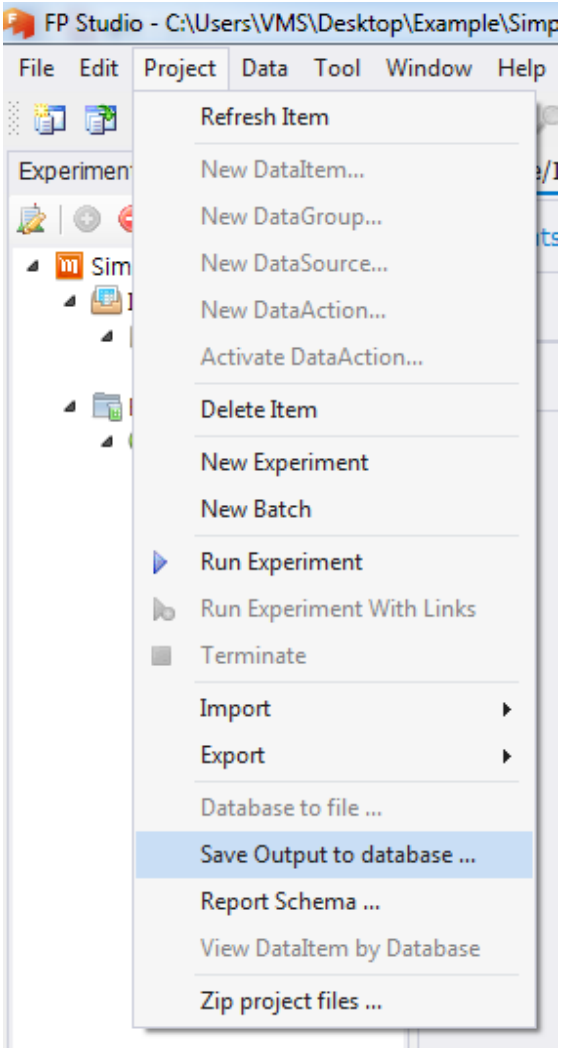
Fix Data

Drag a column header here to group by that column

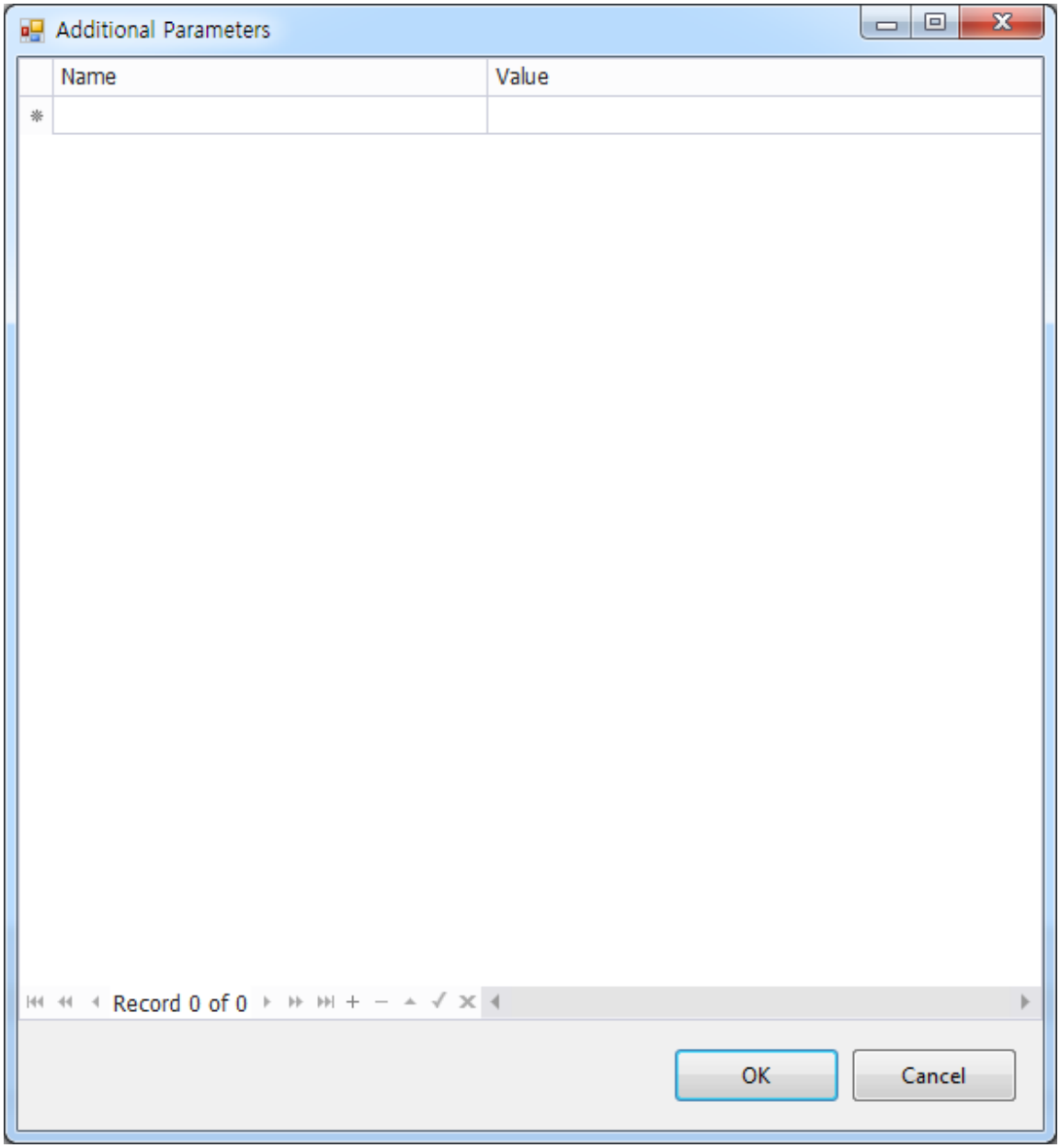
ITEM_GROUP	ITEM_ID	ITEM_NAME	ITEM_PRICE
Furniture	F038	식탁 03	1600000
Furniture	F042	식탁 07	1350000
Furniture	F041	식탁 06	1200000
Furniture	F040	식탁 05	1050000
Furniture	F039	식탁 04	900000
Furniture	F025	쇼파 10	750000
Furniture	F024	쇼파 09	700000
Furniture	F011	책장 05	650000
Furniture	F023	쇼파 08	650000
Furniture	F010	책장 04	640000
Furniture	F009	책장 03	630000
Furniture	F008	책장 02	620000
Furniture	F035	책상 10	620000
Furniture	F007	책장 01	610000
Book	B021	책 21	600000
Furniture	F006	의자 06	600000

Record 6 of 26

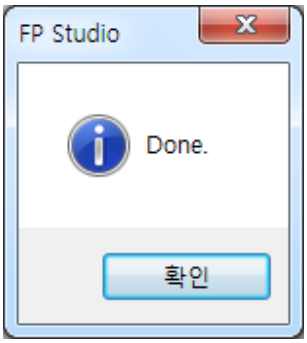
7. MOZART Studio 의 **[Project > Save Output to database…]** 메뉴를 실행하여 실제 Data 저장에 문제가 없는지 확인합니다.



8. Additional Parameters 입력창이 열리고, 추가로 입력할 Parameter 가 있을 경우 Parameter Name 과 Value 를 입력한 후 **[OK]** 버튼을 클릭합니다.
9. 본 예제에서는 추가로 입력할 Parameter 가 없으므로 추가 입력사항 없이 **[OK]** 버튼을 클릭합니다.



10. 등록된 Output Data Table 에 Sorting 된 Data 가 저장되고 있다는 진행창이 뜬 후, “Done” 창이 나타나면 **[확인]** 버튼을 클릭합니다.



11. 결과를 확인합니다.

워크시트 질의 작성기

```
1 SELECT * FROM TBL_ITEM_MASTER_SELECTED ;
2
```

스크립트 출력 x 질의 결과 x

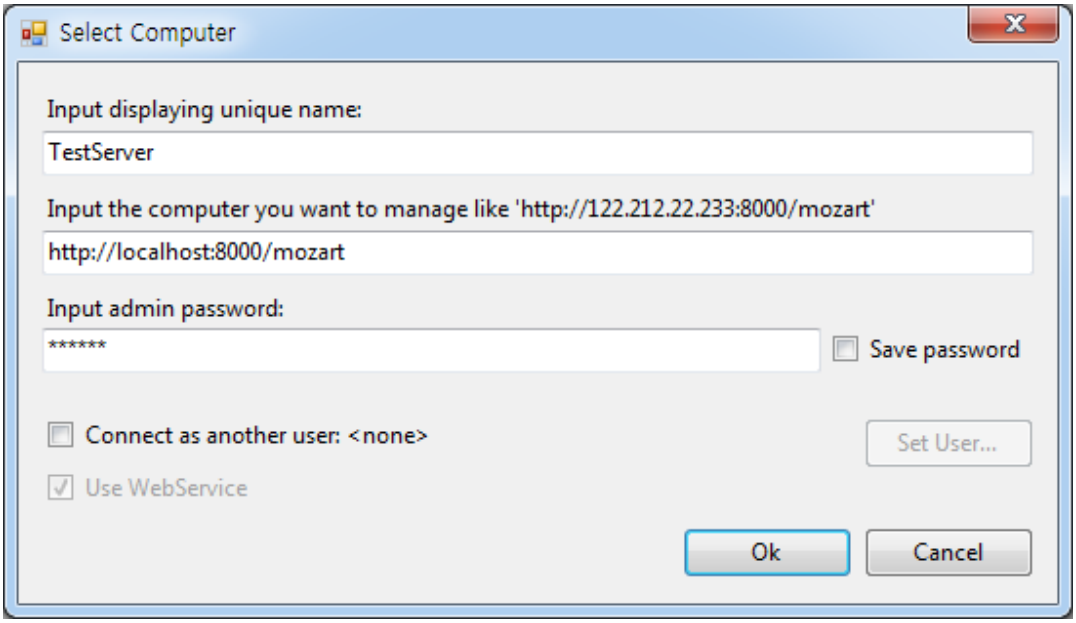
SQL | 인출된 모든 행: 26(0.015초)

	ITEM_GROUP	ITEM_ID	ITEM_NAME	ITEM_PRICE
1	Furniture	F041	식탁 06	1200000
2	Furniture	F040	식탁 05	1050000
3	Furniture	F039	식탁 04	900000
4	Furniture	F025	쇼파 10	750000
5	Furniture	F024	쇼파 09	700000
6	Furniture	F011	책장 05	650000
7	Furniture	F023	쇼파 08	650000
8	Furniture	F010	책장 04	640000
9	Furniture	F009	책장 03	630000
10	Furniture	F008	책장 02	620000
11	Furniture	F035	책상 10	620000
12	Furniture	F007	책장 01	610000
13	Book	B021	책 21	600000
14	Furniture	F006	의자 06	600000
15	Furniture	F022	쇼파 07	600000
16	Furniture	F037	식탁 02	600000
17	Furniture	F005	의자 05	590000
18	Furniture	F034	책상 09	590000

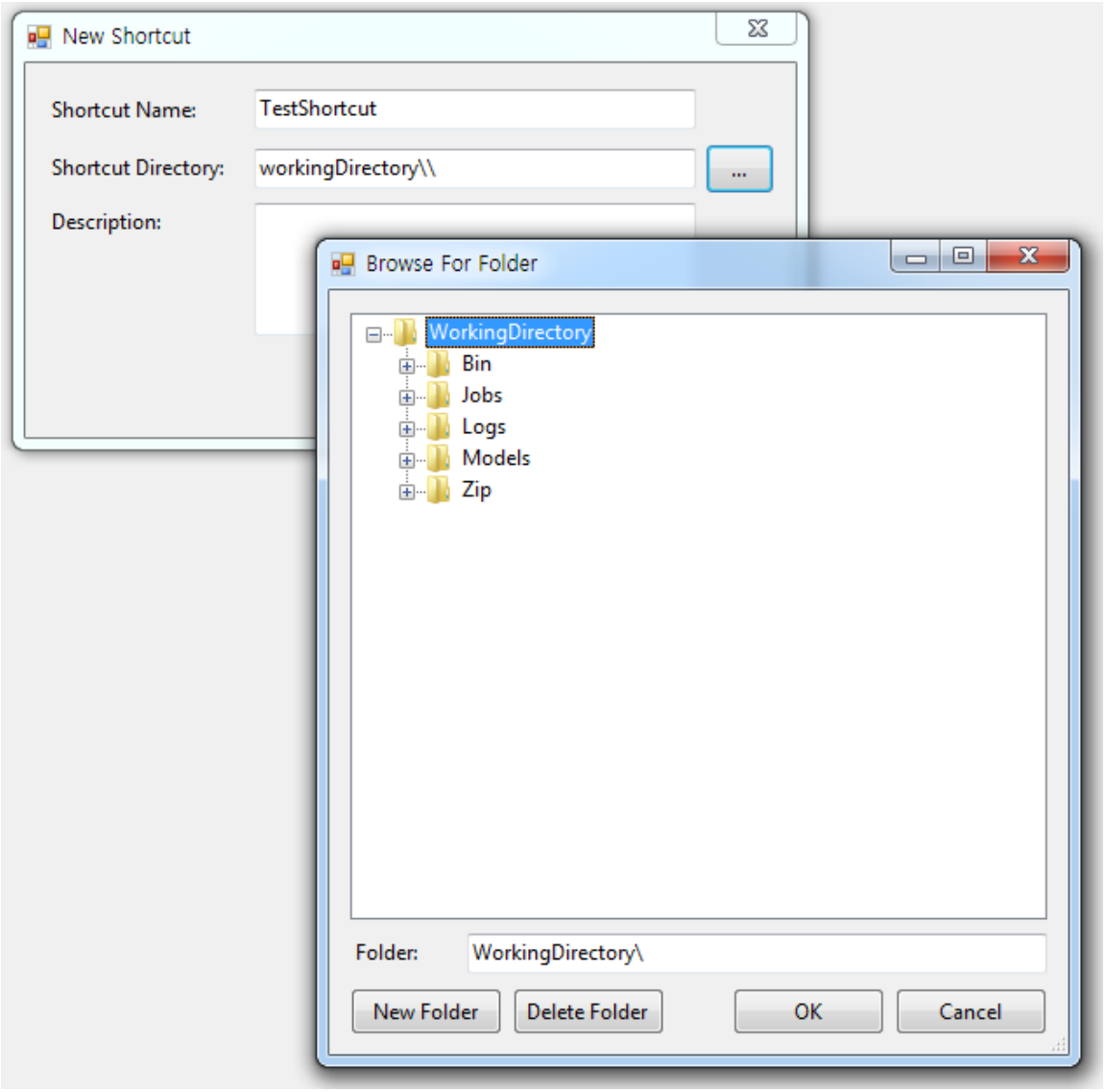
Server Job 생성

개발된 Dll 은 솔루션 폴더 하위의 [obj/Debug] 폴더에 Project 와 동일한 이름으로 생성되어 있으며, Model 은 Generated 폴더 하위에 “.vmodel” 파일로 저장됩니다. Server 에서 개발된 Model 을 적용시키기 위해서는 두 가지 파일을 Server 에 배포한 후 Server 상에서 실행 가능하도록 Job 을 등록해야 합니다. Job 을 등록하기 전, 먼저 Server 와 Shortcut 을 생성합니다.

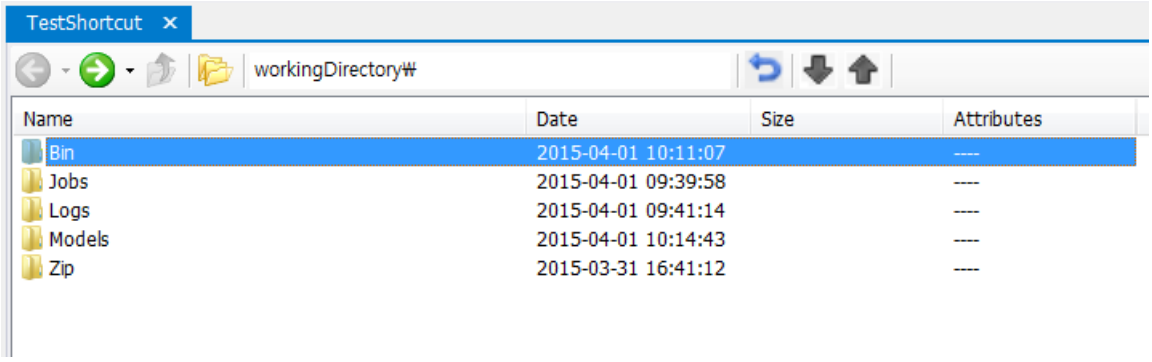
1. MOZART Management Console (MMC) 을 실행시키고 Add Server Conneciton 아이콘을 클릭하여 Server 를 등록합니다. Server 가 설치된 PC 의 IP 와 Server admin 암호를 입력하여 Server 를 등록합니다.



2. Server 접속이 성공하면 Server 에 대한 Tree 노드가 생성됩니다.
3. Server 등록 후, 먼저 필요한 파일을 Server 에 배포하는 작업을 수행합니다. 배포작업을 위해 Server 의 Dll 및 Model 저장 폴더에 대한 Shortcut 을 생성합니다.
4. 생성된 Server 를 선택하고 ShortCuts 아이콘을 오른쪽 마우스 버튼을 클릭하여 **[Add Shortcut]** 팝업메뉴를 실행합니다.
5. New Shortcut 입력창이 나타나면, Shortcut Name 을 입력하고 Shortcut Directory 를 'WorkingDirectory\' 로 지정한 후 **[OK]** 버튼을 클릭합니다.



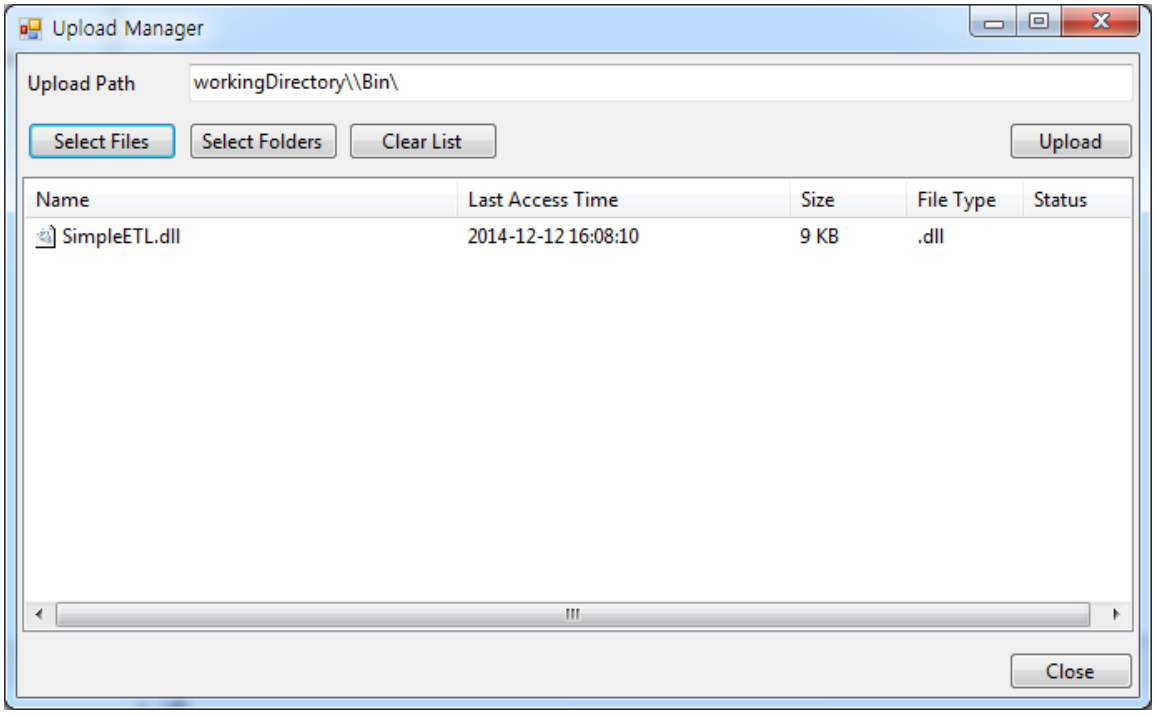
6. Server Tree 노드에 입력한 Shortcut 명으로 Shortcut 이 생성됩니다. 생성된 Shortcut 을 더블 클릭하여 Working Directory 내의 Jobs 폴더에 Dll 파일을 업로드 합니다.



7. Jobs 폴더 내에서 ⬆️버튼을 클릭하면 Upload Manager 창이 나타납니다.

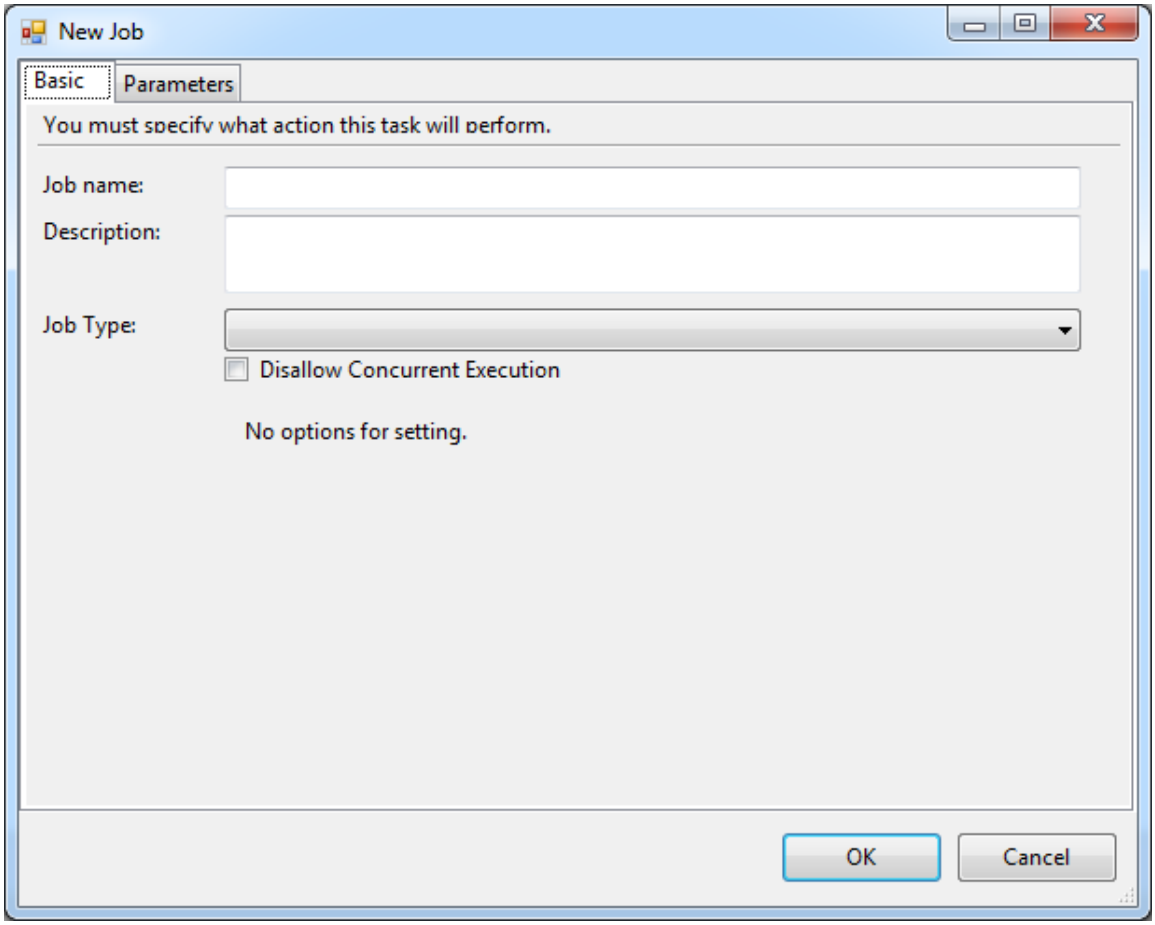
8. Upload Manager 창에서 **[Select Files]** 버튼으로 업로드 대상 파일을 선택하고 리스트에 표시된 상태에서 **[Upload]** 버튼을 클릭하여 업로드를 마무리 합니다.

   데이터 업로드 창

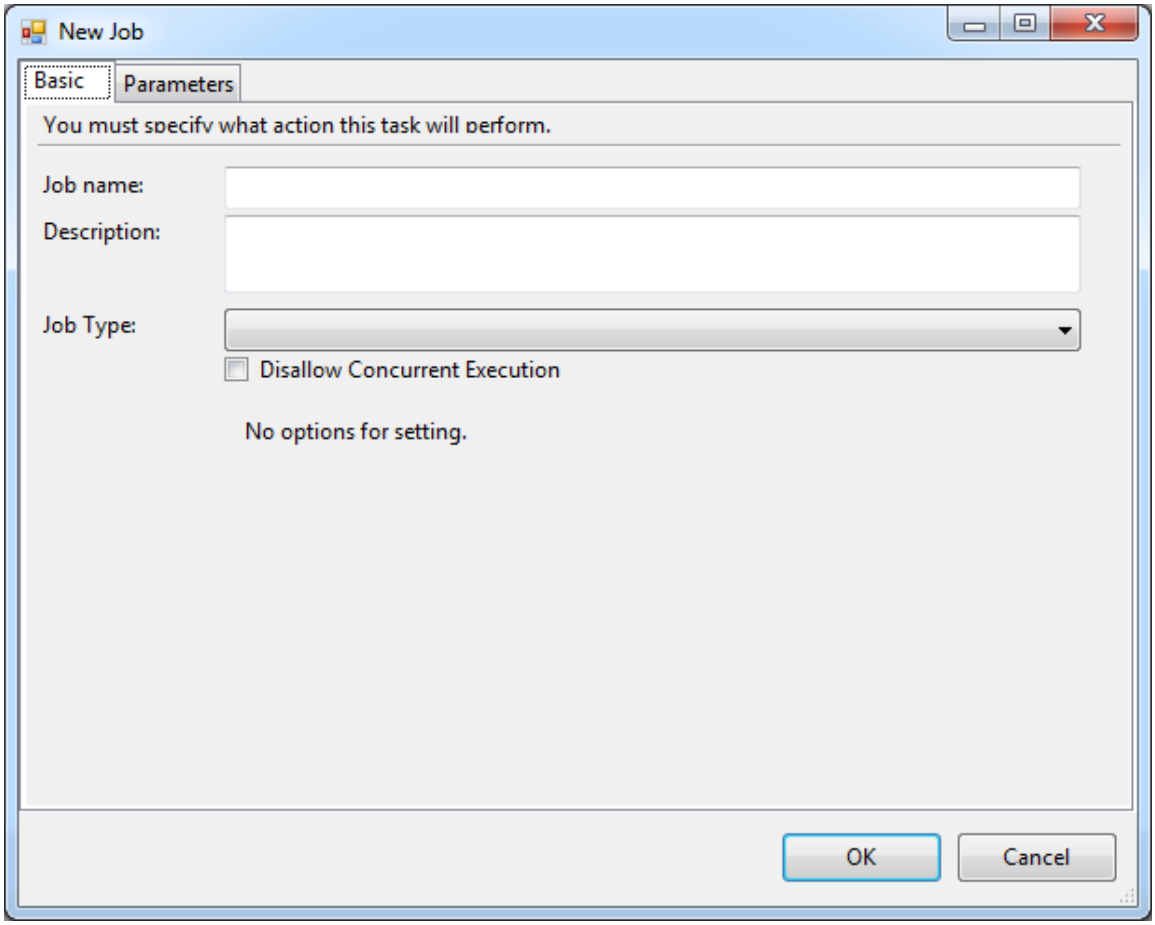


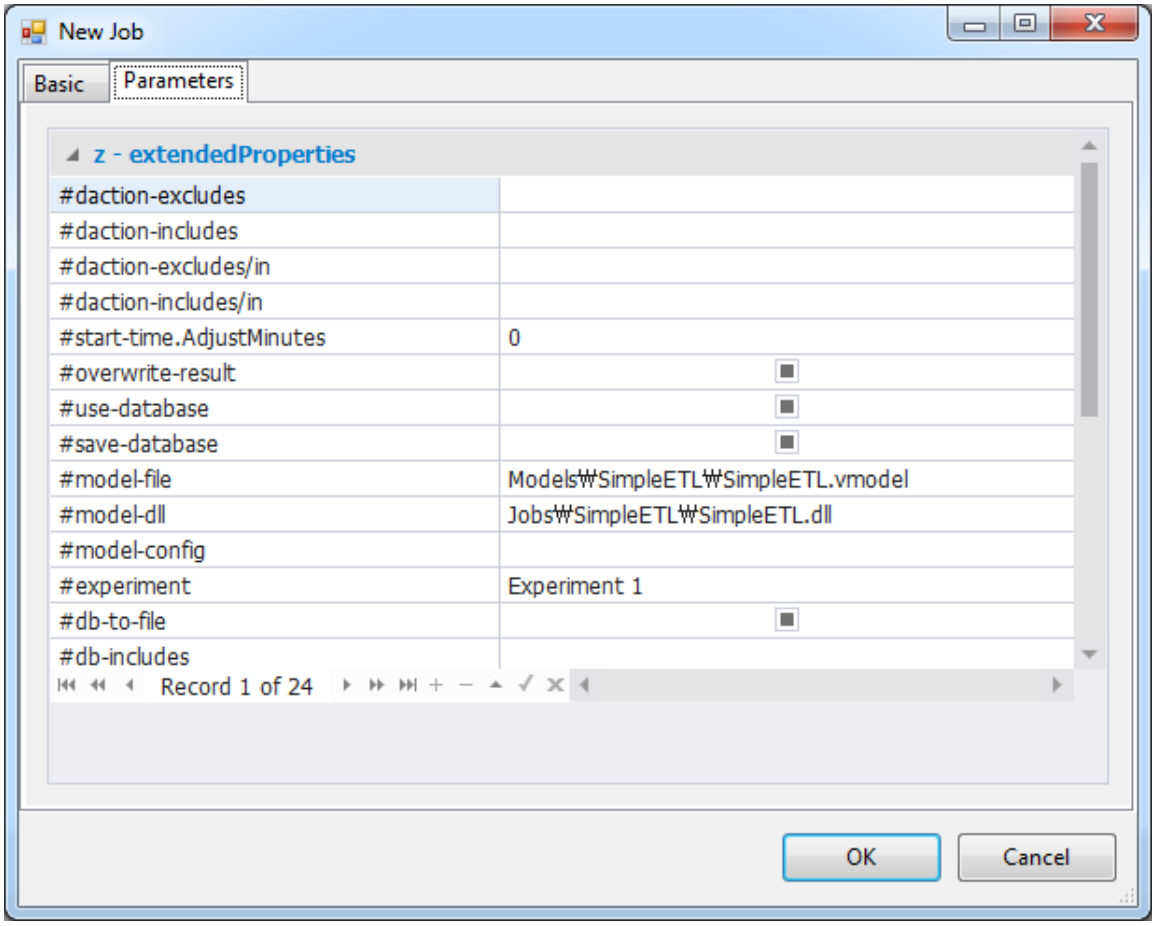
9. 동일한 방식으로 WorkingDirectory 내의 Models 폴더에 Vmodel 파일을 업로드합니다. 등록한 Server 에 실행 대상 파일까지 업로드 되었으면 Job 을 등록합니다.
10. Server 하위의 Jobs 노드를 더블 클릭하거나 오른쪽 마우스 버튼을 클릭하여 **[Open]** 팝업메뉴를 실행합니다.
11. 등록된 Job 리스트를 조회한 후 동일 화면 상단의 **[Add]** 버튼을 통해 Job 을 추가합니다.

    Job 명 및 Job Type 입력



Job Parameters 입력


New Job
Basic
Parameters
You must specify what action this task will perform.
Job name:
Description:
Job Type:
Disallow Concurrent Execution
No options for setting.
OK
Cancel

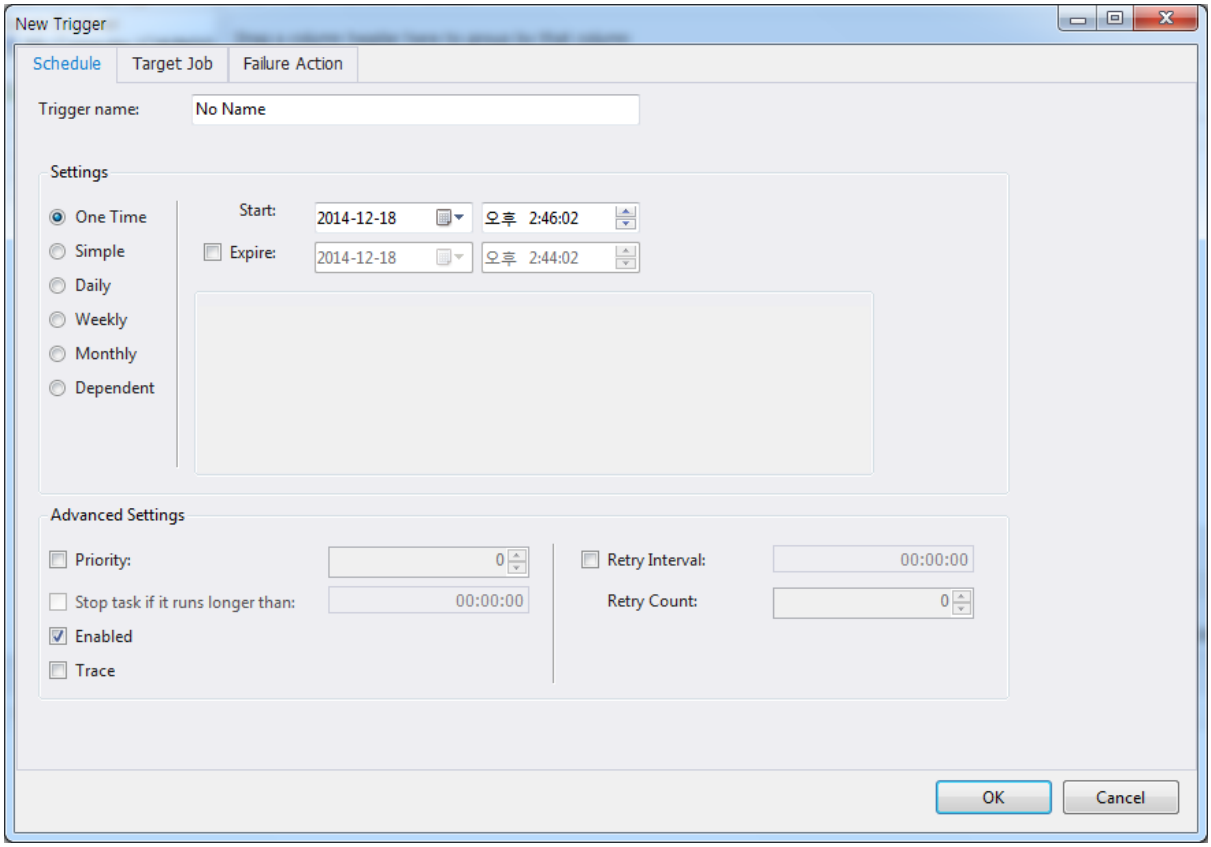


Server 실행 Schedule 생성 및 Log 확인

Job 등록이 끝나면 해당 Job 을 실행하는 Trigger 를 생성하여 Job 을 Server 상에서 구동시키는 작업을 수행합니다.

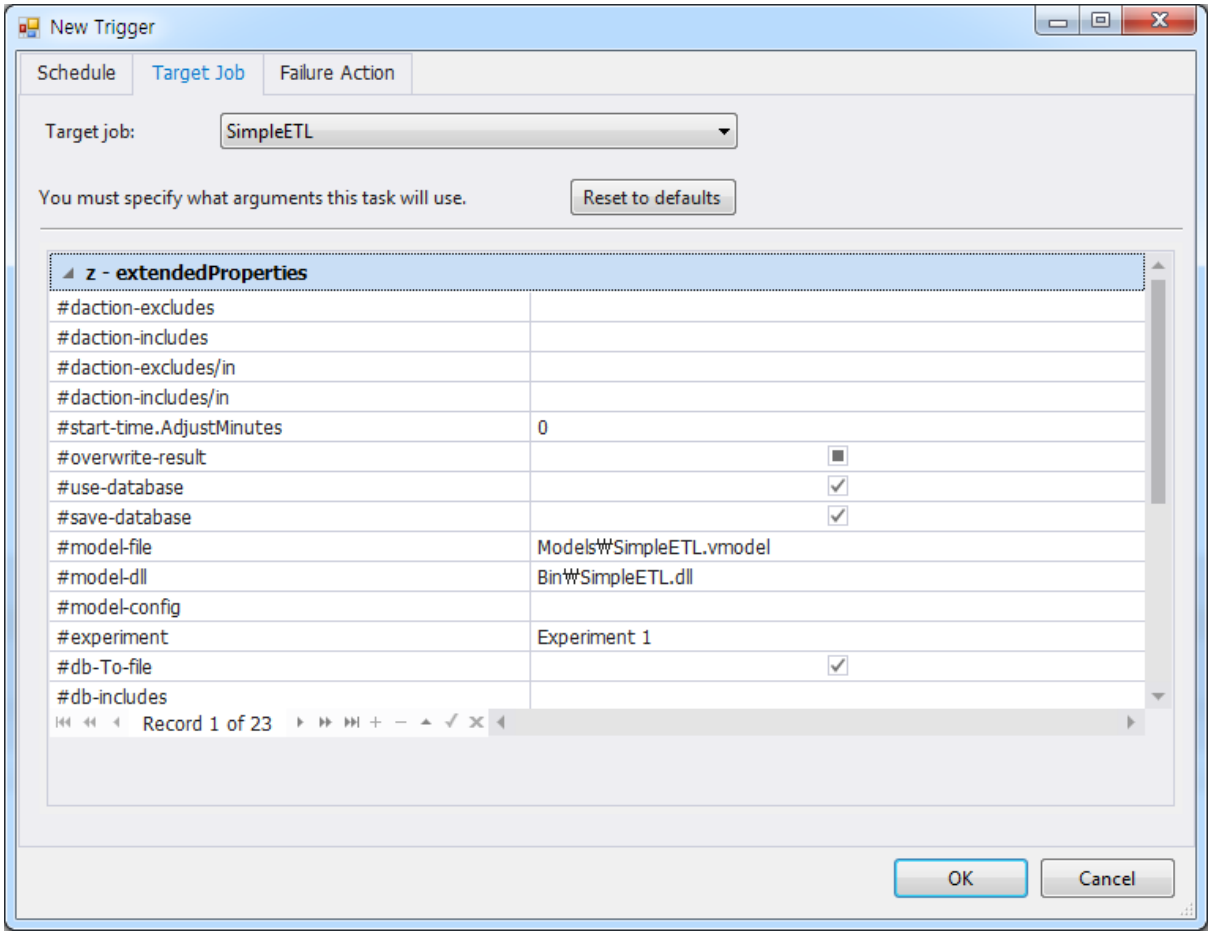
1. Server 하위의 Triggers 노드를 더블 클릭하거나 오른쪽 마우스 버튼을 클릭하여 **[Open]** 팝업메뉴를 실행합니다.
2. 등록된 Trigger 리스트를 조회한 후 동일 화면 상단의 **[Add]** 버튼을 통해 Trigger 를 추가합니다.

Trigger 명 및 Trigger 조건 입력



Trigger 기본 정보 및 실행 조건 입력

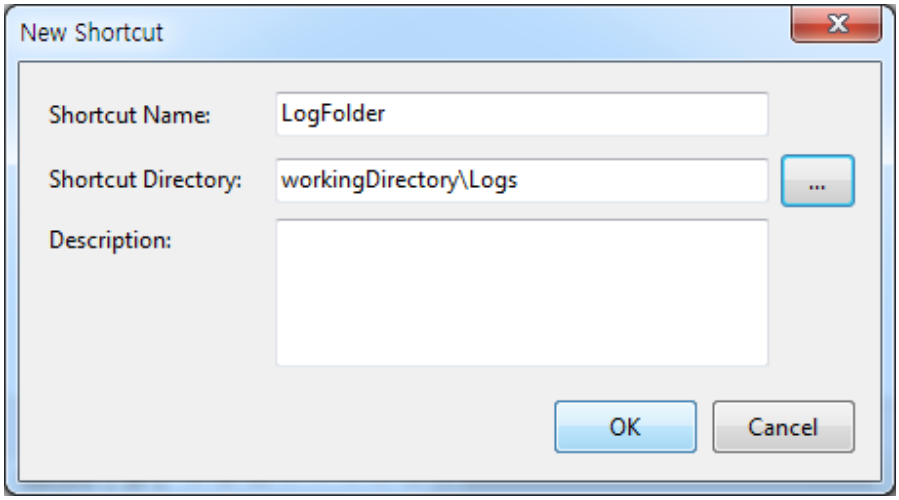
Trigger 의 실행 대상 설정 : 사전에 등록된 Job 을 선택합니다.



실행대상 Job 설정

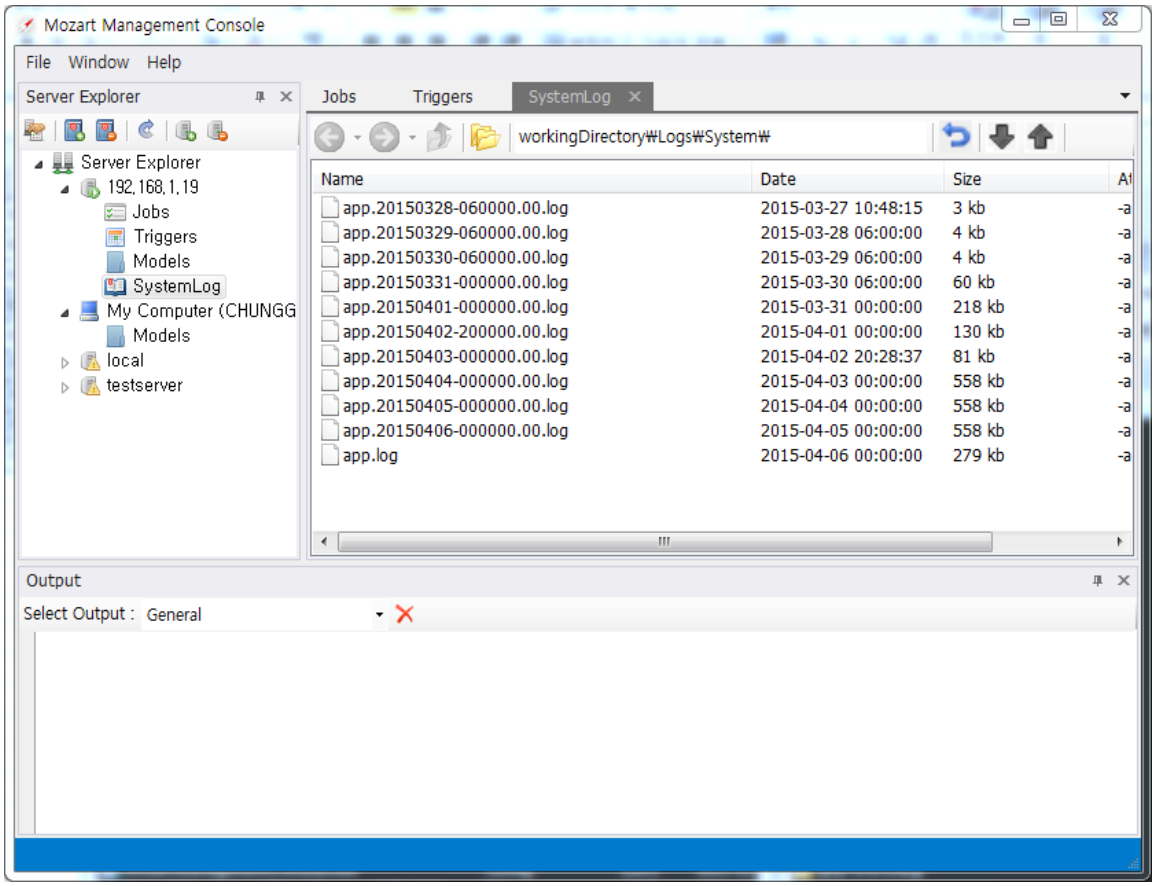
본 화면에서 설정하는 Job Parameters 의 경우 선택한 Target Job 에 사전 설정된 Parameters 값으로 자동 설정되며, 필요 시 조정 가능합니다. 실행 시에는 Trigger 에 설정된 Parameters 가 우선합니다.

1. 모든 입력이 끝나면, 실제 Server 에서 Job 이 구동되는 이력을 확인합니다.
2. Log 확인을 위해 Shortcut으로 Server 의 System Log 폴더를 추가합니다.



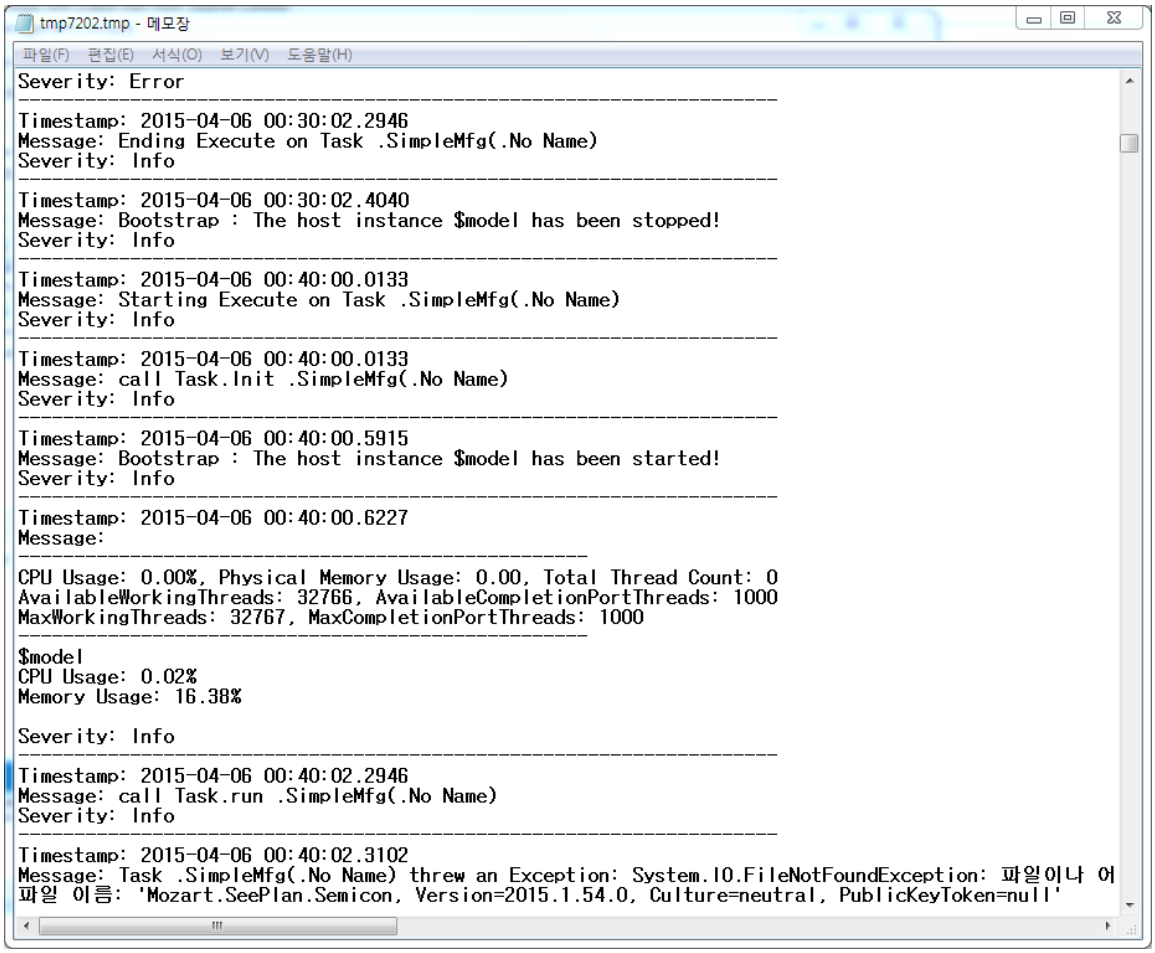
3. **[Working Directory / Logs / System]** 폴더에 app.log 를 확인하여 시스템에서 등록한 Job 들의 전체 실행 여부를 확인합니다. 당일의 Log 는 app.log 파일에 기록되며 일 단위로 파일이 생성됩니다.

Server 전체 job 실행 Log 파일 View



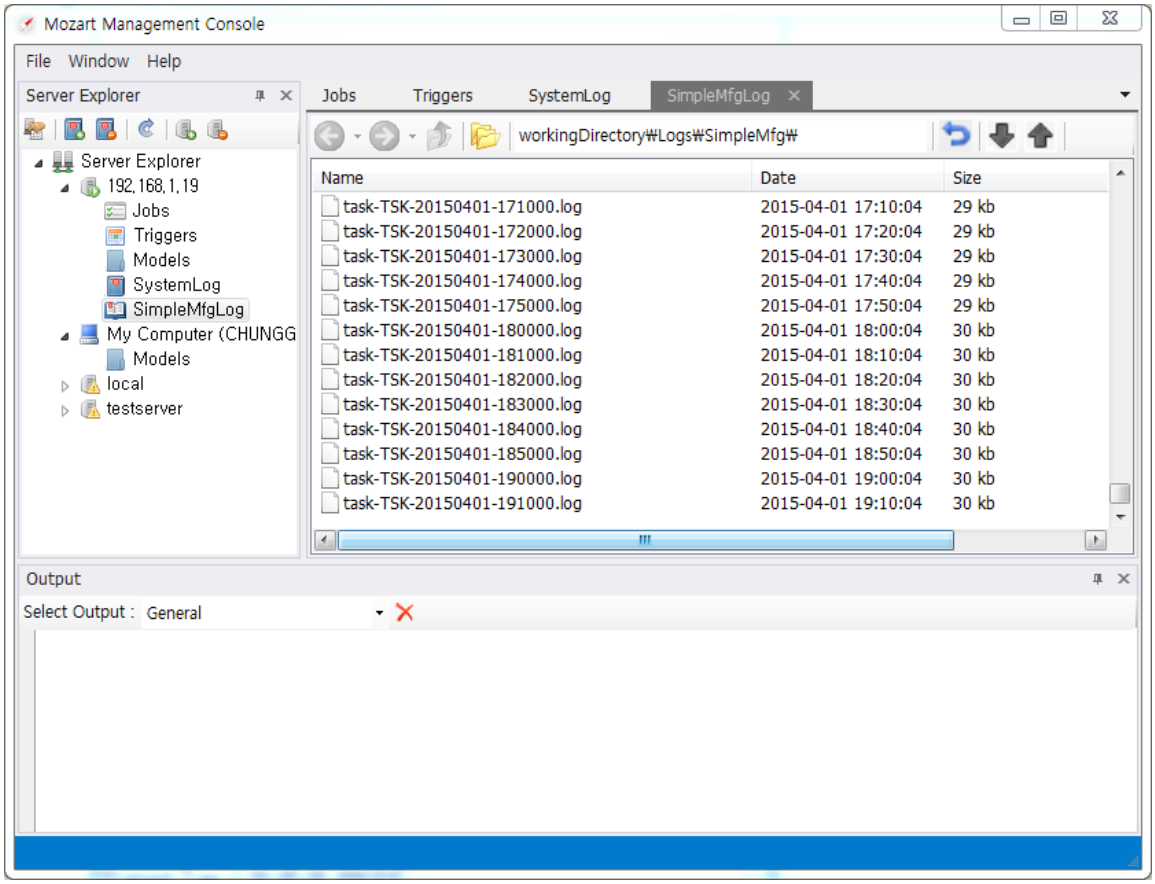
상기 파일을 더블 클릭하면 아래와 같이 시스템의 실행 Log 를 시간순으로 확인할 수 있습니다.

app.log view



4. app.log 외에 개별 실행 Log 는 지정한 Log 폴더에 생성되며, 미지정 시 Logs 폴더 밑에 실행한 Trigger 의 실행 시점별로 생성됩니다.

Task 별 실행 로그 View



마찬가지로 해당 Log 파일을 더블 클릭하면 txt 파일로 실행 Log 확인이 가능합니다.

MOZART IDE

OVERVIEW

MOZART IDE 개요

MOZART IDE (Integrated Development Environment) 는 MOZART Library 와 실행환경을 통해 실행 가능한 실행 Module 을 개발하고 테스트하기 위한 개발환경입니다. MOZART

IDE 는 Visual Studio 의 확장 (Extension) 형태로 제공되므로, IDE 의 사용을 위해서는 Visual Studio 2010 Professional 이상의 환경이 필요합니다. MOZART 개발 Project 는 MOZART Explorer 를 통해 개발 요소와 편집 기능에 접근 가능합니다. 아래는 MOZART Project 의 구성요소를 설명합니다.

1. Model

Model 은 MOZART 개발 Project 의 Logic 수행에 필요한 Inputs, Outputs Data (Schema) 및 실행시점에 Data 를 Logic 에서 사용할 수 있는 객체 형태로 전환하거나 Logic 의 수행결과를 Data File, Database 에 저장하기 위한 Query 와 설정정보를 저장하고 있는 정보파일입니다. 또한 Project 수행 방식이나 옵션을 위해 입력 받을 수 있는 Arguments 정보도 포함합니다. MOZART IDE 를 통해 개발되는 Project 의 경우, 대부분 다양한 Input Data 를 사용하고 여러 Data Source 를 사용하게 되는 경우가 많으므로 Model 을 통해 이러한 정보를 일관되게 관리하고 모델링 할 수 있습니다.

2. My Objects

MOZART 는 대량 Data 의 빠른 처리를 목표로 함에 따라 필요한 Data 를 Logic 처리에 용이한 형태의 Data Model 로 변환하여 Memory 에 로드한 상태에서 Logic 처리를 시작합니다. Logic 처리를 담당하는 Main Module 과 Execution Module 의 실행 시 사용하게 될 Data Model 클래스를 새로 정의하거나, Library 에서 제공하는 Data Model 을 재정의하여 사용하기 위한 기능입니다.

3. My Methods

MOZART Main Module 및 Execution Module 에서 공용으로 사용할 수 있는 Logic 처리 함수를 정의하기 위한 기능입니다.

4. Main Module

MOZART IDE 를 통해 개발된 각 실행 Module 의 Configuration 정보 Setting 및 실행 Module 의 실행을 위한 Setup 및 실행 제어를 위한 기능입니다.

5. Execution Module

MOZART IDE 의 실행 Module 로 MOZART Library 를 통해 제공됩니다. SEEPLAN Library 에는 MOZART 기반의 Plan & Scheduling 시스템 구성을 할 수 있는 Pegging, Simulation Module 이 기본 제공됩니다. 이러한 실행 Module 은 IDE 를 통해 추가할 수 있으며, 추가된 Module 은 Main Module 을 통해 실행 제어가 가능합니다.

Feature Extension 개념

SEEPLAN Library 는 MOZART 실행 프레임워크에 기반하여 제조업의 생산 Planning 및 Scheduling 을 수행하기 위한 Simulation Model 과 Pegging Model 을 제공합니다. Library 로 제공되는 이러한 Model 은 각각 **Core Layer** 와 **Control Layer** 로 구성됩니다.

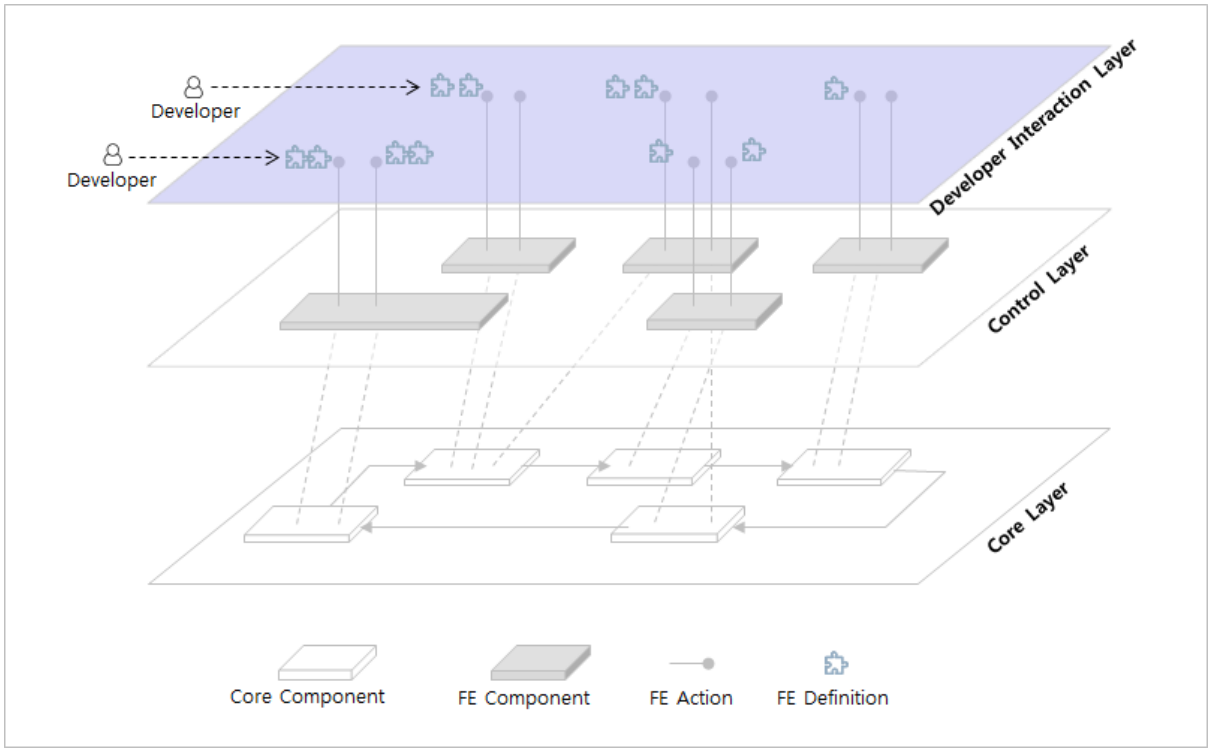
Core Layer 는 Simulation, Pegging 을 구성하는 기본적인 단위 Model 의 집합입니다. 각 단위 Model 은 정해진 역할이 있으며 이러한 역할을 수행하기 위한 기능이 정의됩니다. 그리고 전체 Model 의 실행을 위해 단위 Model 간의 연계관계나 상호 작용 방식이 정의되어 있습니다.

그러나 Core Layer 를 구성하는 단위 Model 의 기능은 실제 적용 시 다양한 방식으로 실행될 수 있으며 적용대상에 따라 상호작용 방식도 달라질 수 있습니다. 예를 들어 RTD(Real Time Dispatcher) 가 대기 중인 작업물의 우선순위를 평가하여 평가순위에 따라 작업물을 선택하면 장비에서는 작업물을 로딩하고 프로세싱하는 역할을 수행합니다. 필요에 따라 Setup 을 수행하기도 합니다. RTD, 장비를 단위 Model 이라고 하면 이들의 역할과 기능은 동일하지만 모델링 하는 대상에 따라 재공의 우선순위를 평가하는 방식, 프로세싱에 소요되는 시간 산출 방식, Setup 판별 방식, Setup 처리 방식은 달라지게 됩니다.

이러한 특성에 따라 MOZART Model Library 는 Core Layer 의 상위단에 Core Component 의 기능, 상호작용 방식을 재정의할 수 있는 **Control Layer** 를 두어 Control Logic 만을 분리했습니다. MOZART 의 Model 기반 개발방식은 Control Layer 를 통한 Logic 개발이 중심이 됩니다. 따라서 Simulation 및 Pegging 의 전체 동작 방식과 Control 계층을 잘 이해하고 있어야 합니다.

Control Layer 를 통해 노출되는 재정의 가능한 Control 지점을 **feature extention Action (FEAction)** 이라 하며, 이들 FEAction 을 Control 대상이나 Logic 의 유사성을 기준으로 Grouping 하여 **feature extension Component(FEComponent)** 를 구성합니다.

아래 그림은 SEEPLAN Library 의 구성과 Module Project 의 개념을 설명합니다.

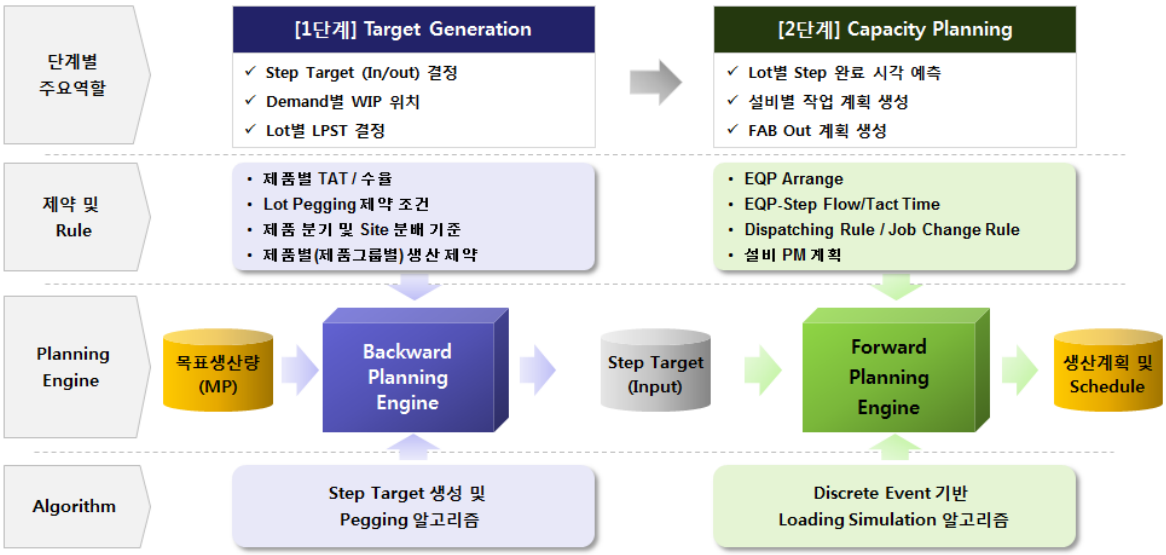


개발자들은 Library 의 FEComponent 를 통해 노출된 FEAction 을 구현함으로써 Core 계층의 Component 를 제어할 수 있게 됩니다. Simulation 및 Pegging Model Project 는 이러한 기본적인 FE Model 외에 각 Model 의 특성에 맞는 통계, Event 처리 등의 특화된 모델링 요소를 추가로 제공합니다.

Feature Extension 기반 Module 개발 방법

Execution 모듈 라이브러리

MOZART Library 에서 제공하는 Execution Module 은 Backward Planning 을 담당하는 Pegging Module 과 Forward Planning 을 위한 Simulation Module 두 가지가 있습니다. Planning & Scheduling 시스템 구축을 위해서는 MOZART 의 두 가지 Module 을 용도에 맞게 추가해서 Logic 을 구성할 수 있습니다. 일반적으로 Factory level 의 Planning 이나 Schedule 을 수행하는 경우 1차적으로 Pegging 을 통해 공정별 생산목표를 설정하고 각 재공의 진도를 파악하는 작업을 수행합니다. Demand 를 충족하기 위한 공정별 생산목표에 기반하여 Simulation 을 통해 각 장비의 작업계획 및 생산계획 (Plan) 을 수립하게 됩니다. 아래 그림은 Planning 의 단계별 MOZART Engine 의 구성과 역할을 나타냅니다.



Pegging Module

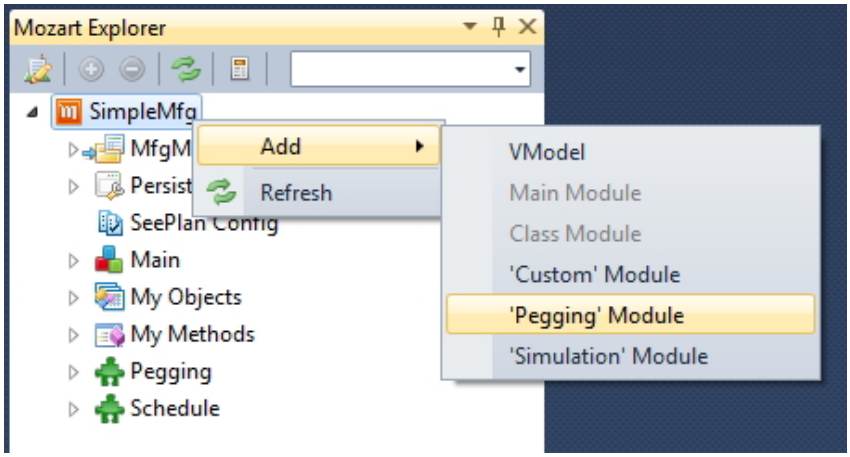
Pegging Logic 개발을 위한 Pegging Library 및 Rule 편집기능, Stage 편집기능을 제공하는 실행가능한 Module 입니다. Main Module 에 의해 구동되며, 실행우선순위는 기본적으로 Simulation 보다 우선합니다. 즉, Pegging Module 과 Simulation Module 을 Project 에 추가한 경우 Pegging Module 가 먼저 실행됩니다.

Simulation Module

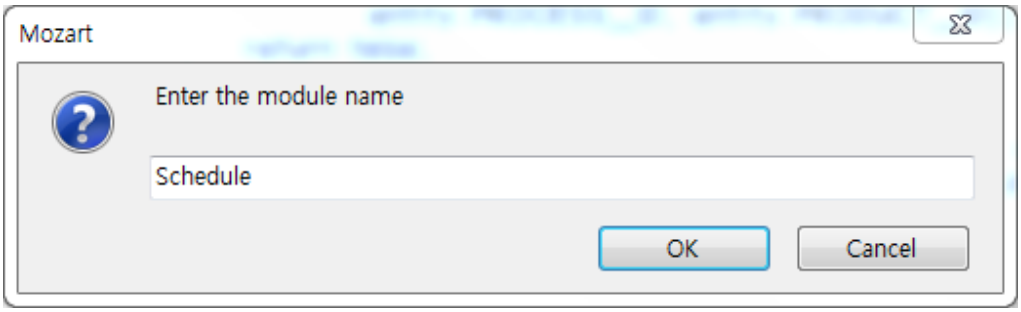
Schedule Logic 개발을 위한 Simulation Library 및 통계수집, Dispatching Weight, 사용자 Event 처리 기능을 제공하는 MOZART Execution Module 입니다. Main Module 에 의해 구동되며, 실행우선순위는 Pegging Module 의 후순위입니다.

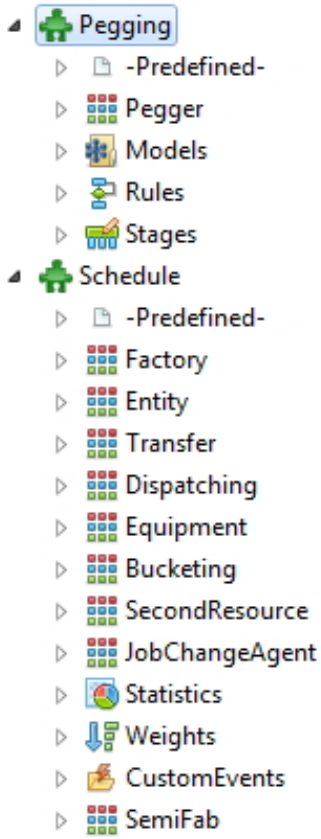
Execution Module 추가하기

1. Execution Module 을 추가하기 위해서는 MOZART Project 의 최상위 노드에서 **[Add > 'Pegging' Module or 'Simulation' Module]** 팝업메뉴를 실행합니다.



2. Module 명을 입력하면 Project Tree 에 입력한 명칭의 Module 이 추가됩니다.
3. 동일 Module 이 복수 개 생성될 수 있습니다.



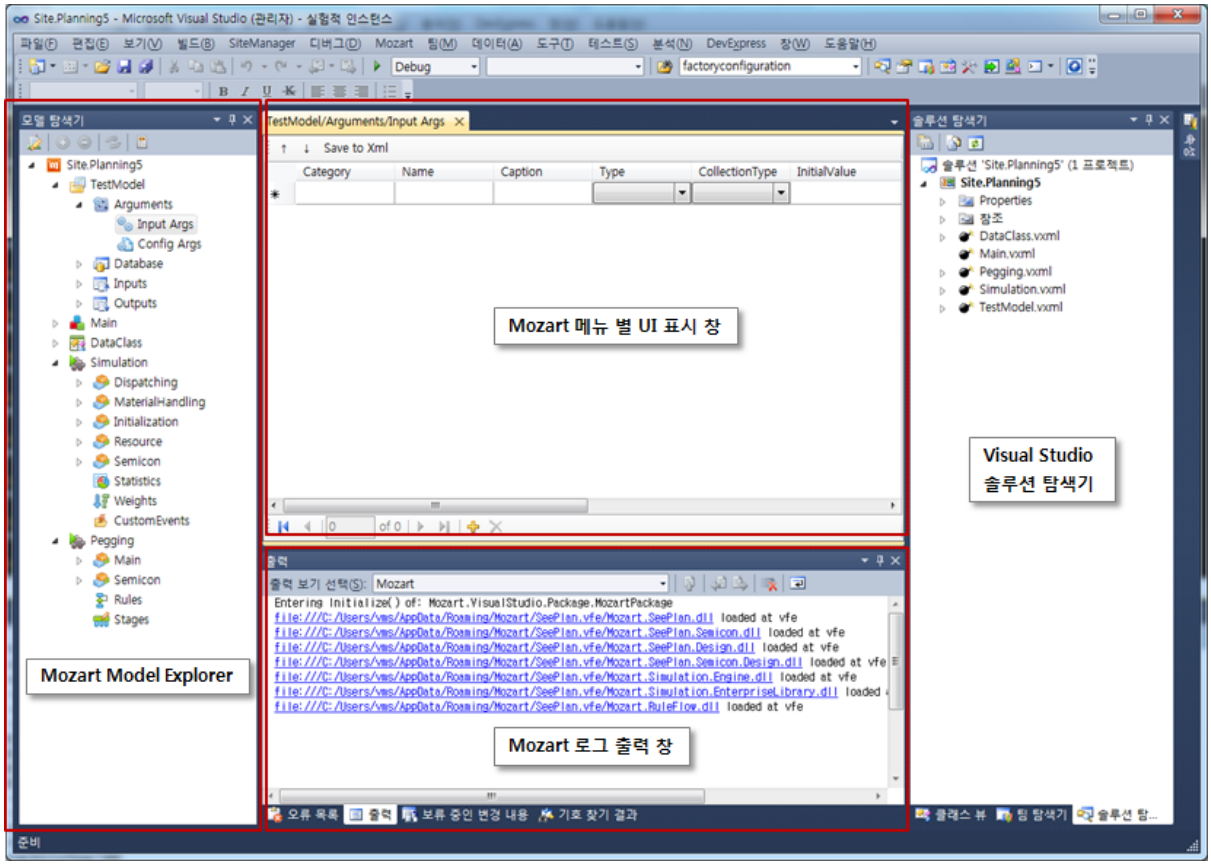


4. Execution Module 추가 시 Solution Explorer 의 Module 명과 동일한 이름의 "Module 명.vxml" 파일이 생성됩니다. 이 파일은 Module 의 정보를 저장하는 파일로 Module 을 삭제하고자 할 때는 MOZART Explorer 가 아닌 vxml 파일을 삭제하면 해당 Module 이 MOZART Explorer 에서도 삭제됩니다.

User Interface

MOZART IDE 는 Visual Studio 의 Package 형태로 공급됨에 따라 Visual Studio 의 기본적인 창 구성과 동일합니다. 따라서 MOZART 로 생성된 Project 의 경우 Visual Studio 의 Solution Explorer 를 통해 Project 파일을 조회할 수 있습니다. 그러나 IDE 의 기능을 사용하기 위해서는 MOZART Explorer 를 사용합니다. MOZART Explorer 는 Solution 을 연 상태에서 Solution Explorer 에서 표시되는 .vxml 파일을 더블 클릭하거나, **[Mozart/Mozart Explorer]** 메뉴를 통해 열 수 있으며, Visual Studio 의 다음 Docking Window 처럼 화면의 원하는 곳에 Docking 가능합니다.

MOZART Explorer 를 통해 사용자는 Model(Vmodel) 편집기능, Custom Data 클래스 편집기능, Custom Method 편집기능, Main Module 개발 기능, MOZART Library 기반 Planning & Scheduling Module 개발 기능을 사용할 수 있습니다. 각 기능에 대한 UI 는 기능에 따라 Visual Studio 의 코드 Editor 가 열리는 부분에서 열리거나 별도의 Dialog 형태로 제공됩니다.

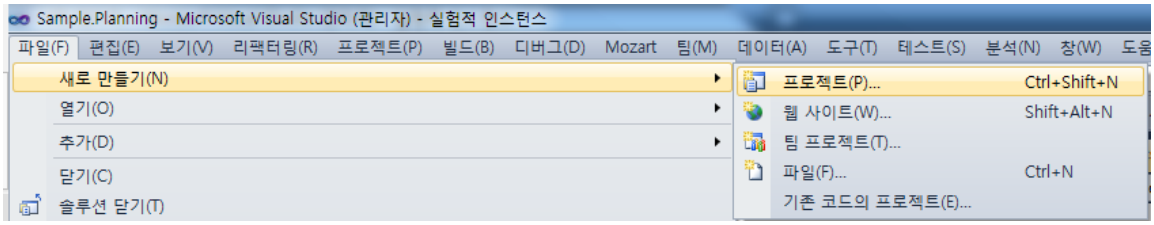


MOZART PROJECT

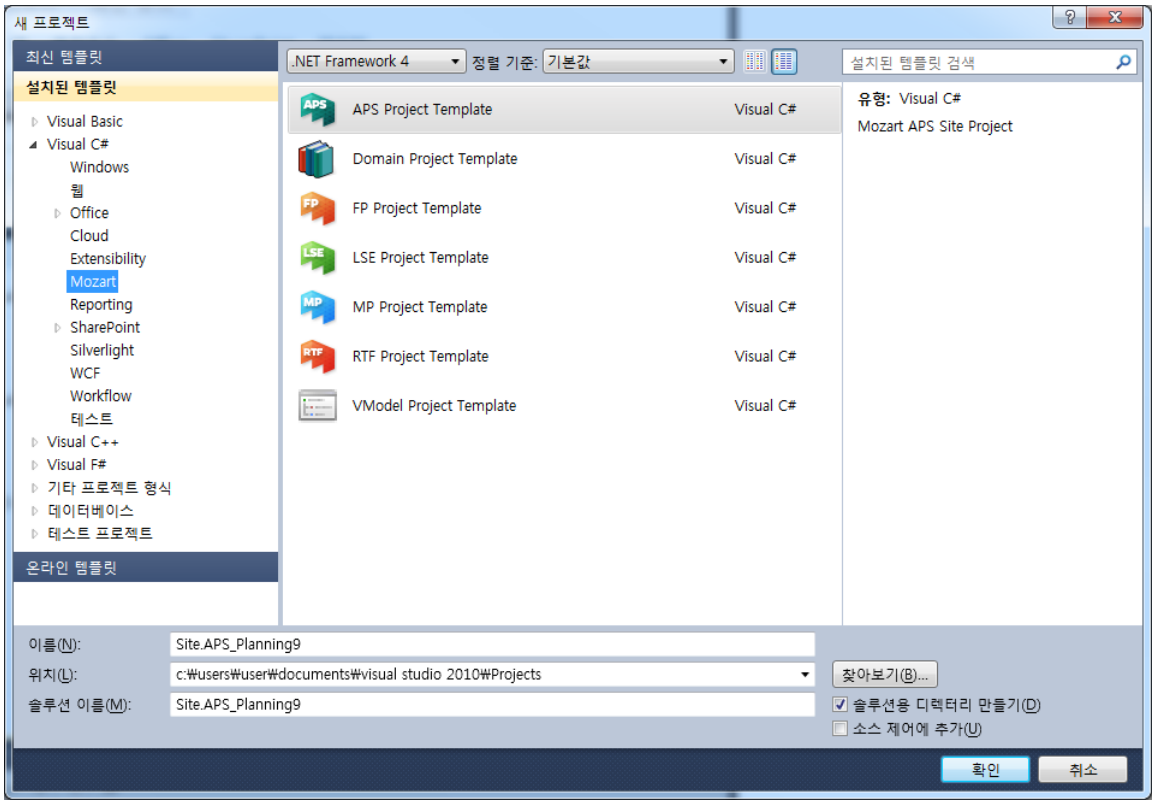
MOZART 프로젝트 생성 방법

Mozart IDE를 설치하면, Visual Studio 기능을 통해 Mozart Proejct를 생성할 수 있습니다.

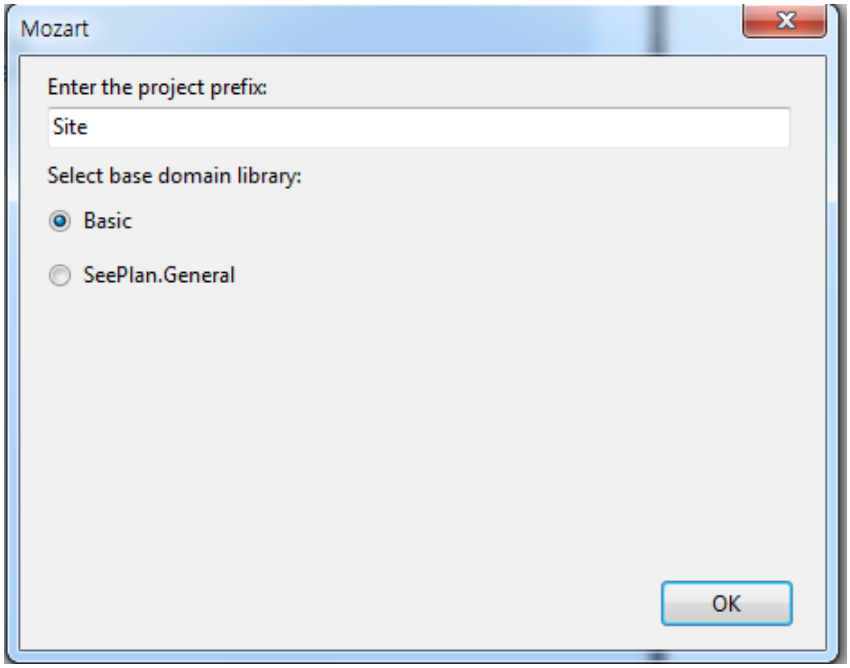
1. Mozart Project 는 **[파일] ->[새로 만들기] -> [프로젝트]** 를 통해 새로운 프로젝트를 생성할 수 있습니다.



2. Mozart 는 C# 기반으로 프로젝트를 구성하기 때문에 설치된 템플릿 -> Visual C# ->Mozart 순서로 찾아가면, 각 License에 맞는 Template을 찾아 구성하실 수 있습니다.



3. Template 을 설정하고 확인을 누른 후, 사용할 Domain Library 를 선택합니다. 설치된 Domain Library가 목록이 나타나며, Project를 구성할 때 사용할 Prefix 도 선택할 수 있습니다.



4. 확인을 누르면 아래와 같이 프로젝트가 생성된 것을 확인하실 수 있습니다.

프로젝트 파일 구성

MOZART IDE 를 통해 자동 생성되는 File 의 종류 및 File 구성 방식을 설명합니다. Project 를 구성하는 Module 에 대한 저장파일은 일반적으로 Project Root Folder 에 생성되며, Model 설정을 통해 자동으로 생성되는 파일은 **[Root / Generated]** 폴더에 생성됩니다.

파일명	설명
./Main.vxml	MOZART Explorer 의 Main Module 에 대한 저장 파일입니다. Configuration 설정 내용과 Main Control 의 구현 내용을 저장합니다. 파일 형식은 XML 을 따르며 항목별 저장방식은 ***Main.vxml 파일형식*** 을 따릅니다.
./Generated/Main.0.cs	Main Module 에서 설정한 Configuration 과 Main Control 에 대한 소스가 자동 생성되는 파일입니다. Model 수정 후 저장시점 혹은 **[Execute Tool]** 메뉴에 의해 생성됩니다.

파일명	설명

파일명	설명
./Model 명.vxml	MOZART Explorer 에서 추가된 MOZART Model 에 대한 저장 파일입니다. Arguments, Inputs, Outputs, Database, Persist Config 의 내용을 모두 저장합니다. 파일 형식은 XML 을 따르며 항목별 저장방식은 ***Model.vxml 형식***형식을 따릅니다.
./Generated/Model 명.0.cs	Model 의 Arguments 설정내용과 InputMart, OutputMart, TempMart 클래스를 정의한 자동 생성파일입니다. Model 의 Inputs, Outputs 및 Arguments 수정 시 변경됩니다.
./Generated/Model 명.Inputs.cs	Model 의 Inputs 에 정의된 DataItem 에 대한 클래스를 정의한 자동 생성파일입니다. Model 의 Inputs 항목 변경 시 자동으로 업데이트 됩니다.
./Generated/Model 명.Persists.cs	Persist Module 에 대한 클래스를 정의한 자동 생성파일입니다. Input/Output Persist Config 에 설정한 내용을 반영하여 클래스가 생성됩니다.
./Generated/Model 명.vmodel	Project 에 포함되지는 않지만 배포 시에 Model 정보를 저장하는 파일이며, MOZART Explorer 에서 Model 변경 후 저장시점 혹은 **[Execute Tool]** 메뉴에 의해 업데이트 됩니다.
./Logic/PersistInputs.cs	Persist Config 의 Input Config 에서 DataItem 별 Action 의 연결함수가 생성되는 파일입니다. Input Config 에서 **+** 버튼을 통해 함수 명을 입력하면 PersistInputs 파일에 Header 를 생성하며, 사용자가 이 파일에 내용을 구현합니다.
./Logic/PersistOutputs.cs	Persist Config 의 Output Config 에서 DataItem 별 Action 의 연결함수가 생성되는 파일입니다.

파일명	설명
./My Objects.vxml	MOZART Explorer 에서 My Objects 에 대한 저장 파일입니다. My Objects 에 포함된 Enums, Constants 및 My Objects 의 클래스와 View 정보를 모두 저장합니다. 파일 형식은 XML 을 따르며 항목별 저장방식은 ***MyObject.vxml 형식***을 따릅니다.
./Generated/My Objects.0.cs	My Objects 에서 정의된 Enums, Constants 및 My Objects 에 대한 Collection 클래스에 대한 자동 생성파일입니다. Model 저장시점 혹은 **[Execute Tool]** 메뉴에 의해 업데이트 됩니다.
./Generated/My Objects.DataModel.cs	My Objects 에서 정의된 Data 클래스 파일을 자동 생성한 파일입니다. Model 저장시점 혹은 **[Execute Tool]** 메뉴에 의해 업데이트 됩니다.

파일명	설명
./[Simulation 모듈명].vxml	MOZART Explorer 에서 추가된 Simulation Module 에 대한 저장 파일입니다. Simulation Module 에 추가된 모든 항목에 대한 설정내용을 저장합니다. 파일 형식은 XML 을 따르며 항목별 저장방식은 ***Simulation.vxml 형식***을 따릅니다.

파일명	설명
./Generated/[Simulation 모듈 명].0.cs	Simulation Module 에서 수정된 내용을 반영하도록 Simulation 실행 클래스를 정의한 파일입니다. [Simulation Module 명].vxml 파일이 저장될 때 자동 생성되며, **[Execute Tool]** 메뉴에 의해 업데이트 됩니다.

파일명	설명
./[Pegging Module 명].vxml	MOZART Explorer 에서 추가된 Pegging Module 에 대한 저장 파일입니다. Pegging Module 에 추가된 모든 항목에 대한 설정내용을 저장합니다. 파일 형식은 XML 을 따르며 항목별 저장방식은 ***Pegging.vxml 형식***을 따릅니다.
./Generated/[Pegging Module 명].0.cs	Pegging Module 에서 수정된 내용을 반영하도록 Pegging 실행 클래스를 정의한 파일입니다. [Pegging Module 명].vxml 파일이 저장될 때 자동 생성되며, **[Execute Tool]** 메뉴에 의해 업데이트 됩니다.

VXML 파일 구조

VXMl 파일별 Data 의 저장 구조를 설명합니다. XML 문서의 Keyword 가 Group, Collection 을 나타내는 경우에 보통 Capacity 속성을 정의하는데 이는 하위 집합의 Item 수와 동일하거나 4의 배수로 설정됩니다. Capacity 이상으로 Item 이 설정될 수 없습니다.

Model.VXMl

- `<TargetDefinitioin>` : Model 의 Target 정보
- `<ArgumentArray>` : Model 의 Input Arguments 정보 저장 Section
  - `<ctm:Argumentinfo-Array>`
    - `<ctm:ArgumentInfo>` : 개별 Argument 정보
- `<ConfigArray>` : Model 의 Config Arguments 정보 저장 Section
  - `<ctm:ArgumentInfo-Array>`
    - `<ctm:ArgumentInfo>` : 개별 Config 변수 정보
- `<Inputs>` : Input Data Section

- `<ItemArray>`
  - `<cdd:DataItem-Array>`
    - `<cdd:DataItem>` : DataItem 별 저장 정보
      - `<cdd:MetaInfo>` : Input DataItem 정보 Section
        - `<cdm:DataMetaInfo>` : DataItem 메타정보 헤더
          - `<cdm:Properties>` : DataItem 속성 집합 Section
            - `<cdm:PropertyMeta>` : 개별 속성에 대한 정보
      - `<cdd:ActiveAction>` : Active 설정된 DataAction 정보
        - `<cdd:DataAction>` : DataAction 정보
      - `<cdd:Actions>` : Active 를 제외한 등록된 모든 DataAction 정보 Section
        - `<cdd:DataAction>` : DataAction 정보
- `<Outputs>` : Output Data Section (Inputs 과 동일한 구성)
- `<DataSource>` : Database 등록내용 저장
  - `<Items>`
    - `<Entry>` : 입력된 Data Source 별로 하나씩 생성
      - `<String>`: Data Source 표시 명
      - `<cdd:DataSource>` : Data Source 전체 Section
        - `<cdd:Items>` : 연결문자열 Section
          - `<cdd:DataSourceItem>` : 연결문자열 정보
- `<PersistInput>` : Input Persist Config 설정 정보 저장
  - `<RootItem>` : Input Persist 전체에 대한 설정 정보 저장
    - `<Config>` : Config Argument 의 Data Source Table 설정 정보 저장
    - `<Items>` : GroupItem 그룹 Section
      - `<GroupItem>` : Persist Input Group 에 대한 설정 정보 저장
        - `<Items>` : EntityItem 그룹 Section
          - `<EntityItem>` : 개별 DataItem 에 대한 Persist Config 설정 정보 저장
- `<PersistOutput>` : Output Persist Config 설정 정보 저장

Main.VXMl

- `<FMainMap>` : name = “Main”
- `<Config>` :
  - `<FConfigEntity Name = "페이지명">`
    - `<ConfigValues>`
      - `<csd:Config 클래스명>` : Config 클래스별 속성 설정
- `<Definitions>` : 사용자가 정의한 Definition 및 사전 정의된 Definition 중 Binding 된 Definition 에 대한 정보
  - `<FDefinitionCategory>` : 생성되는 파일 단위로 생성
    - `<FDefinitionEntity>` : 정의된 Definition 별로 생성
- `<Children>` : Definition 이 정의된 FECategory/FEComponent/FEAction 의 정보
  - `<FActionCategory>` : Model Tree에서 표시된 Category 별로 저장
    - `<FActionComponent>` : Category 하위의 Component 단위로 저장
      - `<FActionEntry>` : Definition 이 추가된 경우에만 생성

My Object.VXMl

- `<FDataClassRoot name = "My Objects">`
- `<Constants>`
  - `<FConstantItem name = "Constants">`
    - `<ArgumentArray>`
      - `<ctm:ArgumentInfo-Array>`
        - `<ctm:AgrumentInfo>` : 개별 Constant 정보
- `<Enums>`
  - `<FDataClassFolder : name = "Enums">`
    - `<FDataClassItem IsEnum=true>` : Enum Item 별 Data
      - `<Content>`

        - `<DataClassMeta>`
            - `<Properties>`
                - `<ctd:PropertyMeta>` : 속성 정보
- `<Children>`
    - `<FDataClassFolder>` : My Objects 의 폴더 정보
        - `<Children>`
            - `<FDataClassItem>` : 폴더 내의 Data 클래스 개별 Item 정보
                - `<Content>`
                - `<DataClassMeta>`
                - `<Properties>`
                - `<ctd:PropertyMeta>` : 속성 정보

Simulation.VXMI

- `<FActionMap InstanceName="XXX" Name="Simulation">`
- `<Definitions>` : 사용자가 정의한 Definition 및 사전 정의된 Definition 중 Binding 된 Definition 에 대한 정보
    - `<FDefinitionCategory>`: 생성되는 파일 단위로 생성
        - `<FDefinitionEntry>` : 정의된 Definition 별로 생성
- `<Children>` : FEModel 을 구성하는 Category 및 Simulation 하위 기능별 Entry 항목 구성
    - `<FActionCategory>` : Model Tree에서 표시된 Category 별로 저장
        - `<FActionComponent>` : Category 하위의 Component 단위로 저장
            - `<FActionEntry>` : Definition 이 추가된 경우에만 생성
    - `<FCustomEntry Name = "Statistics">` : 사용자 정의 통계 저장 Section
        - `<csd:StatSheetInstance>` : 사용자 정의 통계 Item 정보
    - `<FCustomEntry Name = "Weights">` : 사용자 정의 Weight Factor Method 저장 Section
        - `<csd:WeightInstance>` : 사용자 정의 Weight Factor Method 정보

- `<FCustomEntry Name = "CustomEvents">` : 사용자 정의 Event 정보
  - `<csd:CustomEventInstance>` : 사용자 정의 Event Item 정보

Pegging.VXMI

- `<FActionMap InstanceName="XXX" Name="Pegging">`
- `<Definitions>` : 사용자가 정의한 Definition 및 사전 정의된 Definition 중 Binding 된 Definition 에 대한 정보
  - `<FDefinitionCategory>`: 생성되는 파일 단위
    - `<FDefinitionEntry>` : 정의된 Definition 별로 생성
- `<Children>` : FEModel 을 구성하는 Category 및 Simulation 하위 기능별 Entry 항목 구성
  - `<FActionCategory>` : Model Tree에서 표시된 Category 별로 저장
    - `<FActionComponent>` : Category 하위의 Component 단위로 저장
      - `<FActionEntry>` : Definition 이 추가된 경우에만 생성
  - `<FCustomEntry Name = "Rules">`
    - `<csd:PredefinedPegRule>` : 사전 정의된 Rule
      - `<FActionEntry>` : 해당 Rule 에 정의된 Action 을 구현하는 경우 생성
    - `<csd:PegRuleInstance>` : 사용자가 정의한 Rule
  - `<FCustomEntry Name = "Models">`
  - `<FCustomEntry Name = "Stages">`
    - `<csd:PegStageInstance>`
      - `<FPredefinedReference>` : 사전 정의된 Rule 을 Stage 에 추가한 경우
      - `<csd:PegRuleReference>` : 사용자 정의한 Rule 을 추가한 경우

Definition 프로파일링

개요

Mozart IDE의 SeePlan 라이브러리에서 제공하는 Default Definition과 사용자가 구현한 함수(FEAction 및 My Methods)에 대한 프로파일링(Profiling) 기능이 있습니다.

Mozart IDE는 Visual Studio의 확장 도구이기 때문에 Visual Studio에서 제공하는 프로파일러(Profiler)를 사용하여 개발한 프로젝트의 성능 분석을 진행 할 수 있습니다. 그러나 Visual Studio의 경우는 외부 호출을 포함 모든 소스코드에 대한 성능 분석을 진행하는 관계로 코드의 양과 복잡도에 비례하게 분석에 소요되는 시간이 증가합니다.

Mozart IDE에서는 주로 사용자가 추가한 모듈(Pegging, Simulation, CBS, 기타 등등)에서 제공하는 구현 지점에 Definition을 구현하기 때문에 프로젝트의 전체 성능 분석보다는 Definition 한해서만 분석이 필요 할 수 있습니다. Mozart IDE의 프로파일러는 이러한 Definition과 필요에 따라 MyMethod에 포함된 사용자 정의 함수에 대해서만 프로파일링 진행합니다.

프로파일링 클래스 구성

Definition에 대한 프로파일링 집계 및 관리하는 클래스는 `Mozart.Task.Execution.Performance.FunctionProfiler` 이며, 주요 함수 및 속성에 대한 정의는 다음과 같습니다.

EnablingProfiling 속성

프로파일링 실행 여부에 대한 값을 가져오거나 설정합니다.

```
public static bool EnableProfiling {get; set;}
```

속성 값

true이면 프로파일링을 실행합니다.

부연설명

임의 시점에 속성에 대한 값을 설정할 수 있으나, Input Arguments에 `enableFunctionProfiling` 을 추가하면 내부적으로 해당 Argument 값에 따라 프로파일링을 실행합니다.

OnEntry(string) 함수

주어진 함수명에 대한 프로파일 집계 시작 지점입니다.

```
public static void OnEntry(string name)
```

매개변수

`name`

데이터 타입: string

정의: 프로파일 집계 대상 함수명입니다.

OnExit(string) 함수

주어진 함수명에 대한 프로파일 집계 종료 시점입니다.

```
public static void OnExit(string name)
```

매개변수

`name`

데이터 타입: string

정의: 프로파일 집계 대상 함수명입니다

Report 함수

프로파일 대상 함수들의 누적 집계 결과를 문자열로 반환합니다.

```
public static string Report()
```

반환

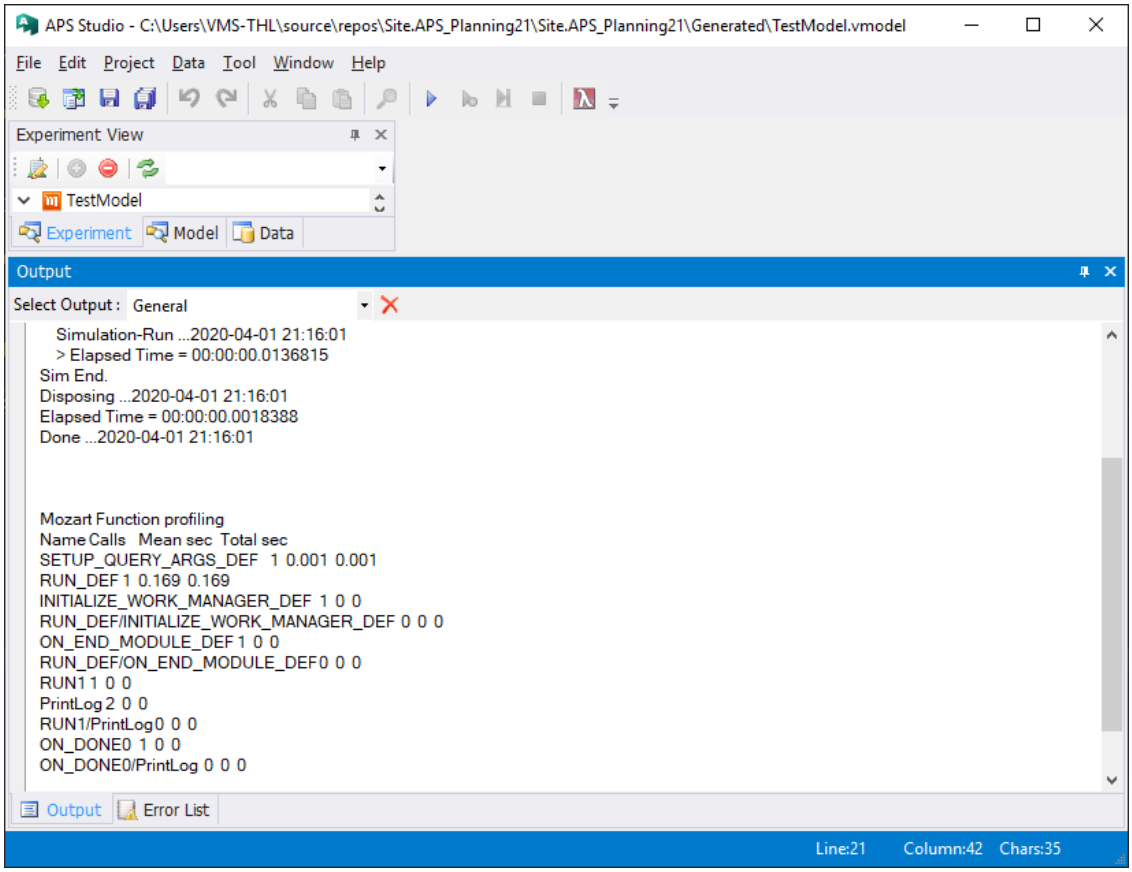
프로파일 대상 함수들의 누적된 집계 결과 문자열입니다.

예제

다음 예제는 Main 컨트롤의 Definition 및 My Methods에서 구현한 함수의 프로파일 집계 결과를 Mozart Studio의 로그창에 출력하는 예제입니다.

```
using Mozart.Task.Execution.Performance;
public void SHUTDOWN0(ModelTask task, ref bool handled)
{
    Logger.MonitorInfo(FunctionProfiler.Report());
}
```

모델 실행이 종료되는 시점인 `Shutdown` FEAction에 위와 같이 코드 한 줄을 작성하면 Mozart Studio의 로그창에 집계 대상 함수들의 프로파일 집계 최종 결과를 확인 할 수 있습니다. 예제에서는 Shutdown에 구현을 했지만 Main 컨트롤의 `OnDone` FEAction에 구현해도 무방합니다.

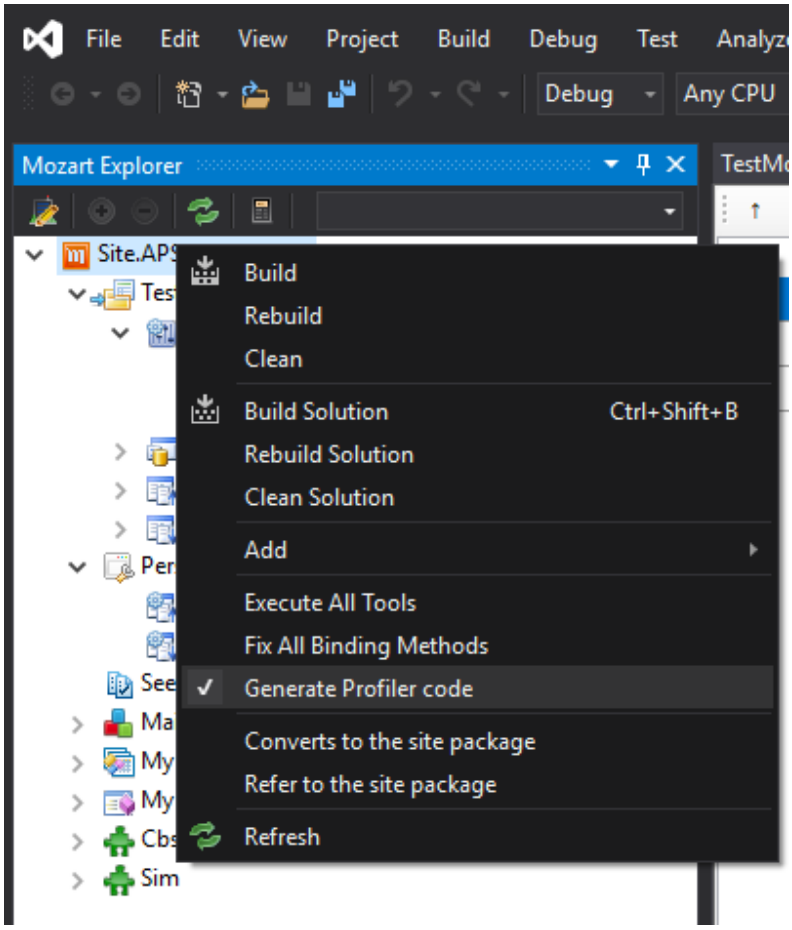


사용 방법

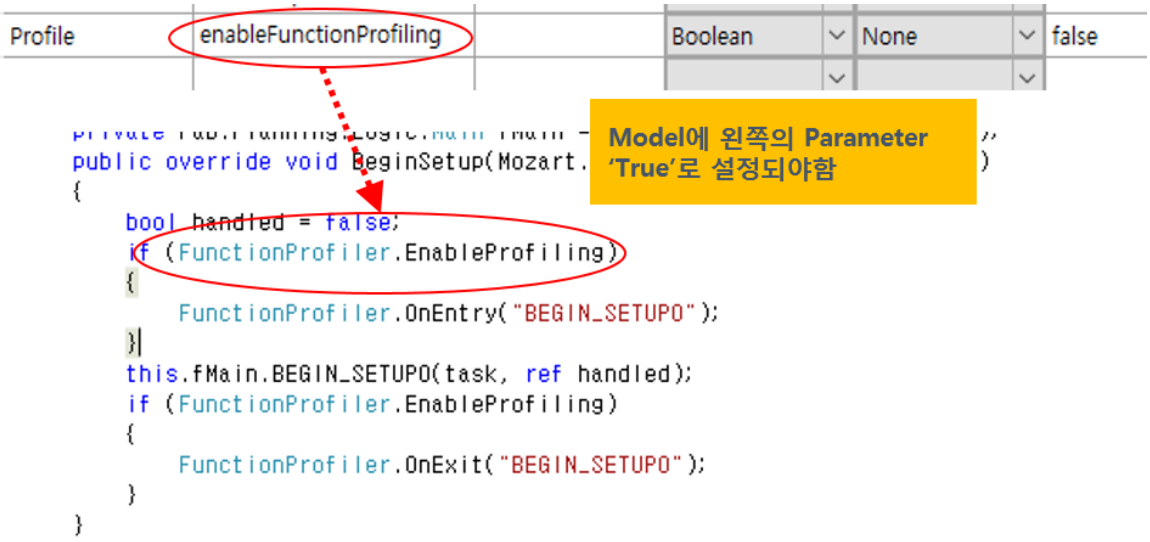
Mozart IDE의 프로파일링 기능 사용 방법은 다음과 같습니다.

1. Mozart Explorer에서 모델의 노드를 펼친 다음 **[Arguments]** > **[Input Args]** 더블클릭하여 Input Arguments 편집 창을 엽니다.
2. 프로파일링 실행 여부를 결정하는 Argument를 다음과 같이 추가합니다. 아래의 Argument명은 고정입니다.
   - **Name**: enableFunctionProfiling
   - **Type**: Boolean

3. 다음은 각 Definition에 대해 프로파일 집계 코드를 자동생성 해주는 기능을 활성화합니다. 기능 활성화는 Mozart Explorer의 Context Menu에서 **[Generate Profiler Code]**를 선택합니다.



위와 같이 설정하면 Generated 폴더에 각 모듈에 해당되는 cs 파일에 정의된 Default Definition 및 사용자 정의 Definition에 대해 Input Arguments 값에 따라 프로파일링을 실행하는 코드가 자동 갱신됩니다.



`enableFunctionProfiling`의 값을 `true` 설정하면 모델이 실행되는 동안 프로파일링을 수행합니다.

Generate Profiler code가 활성화 되어 있는 동안에는 새롭게 추가되는 FEAction에 대해 프로파일링 집계 코드가 포함된 상태에서 Generated 폴더에 있는 cs파일의 코드가 갱신됩니다.

Persist 로직 및 My Methods의 함수 프로파일링 집계 방법

각 모듈의 FEAction의 Definition에 대한 프로파일링뿐만 아니라 Persist의 OnAfterLoad, `OnAction` 그리고 MyMethod에 구현한 사용자 정의 함수에 대해서도 프로파일링이 가능합니다. 설정 방법은 다음과 같습니다.

1. 사용방법의 1~3번까지는 동일하게 진행합니다.
2. 프로파일링 하고자 하는 함수 내부에 집계 시작과 종료에 대한 로직을 추가합니다. 아래는 사용 예제입니다.

```
public void TEMP_METHOD()
{
    if (FunctionProfiler.EnableProfiling)
    {
        FunctionProfiler.OnEntry("TEMP_METHOD")
    }

    //Your code for this method...

    if (FunctionProfiler.EnableProfiling)
    {
        FunctionProfiler.OnExit("TEMP_METHOD")
    }
}
```

위와 같이 프로파일 집계 코드를 추가하면 사용자 정의 함수에 대해서도 프로파일링을 실행합니다. 프로파일 결과는 모델 실행 이력 파일에 자동으로 기록됩니다. 모델 실행 이력 파일의 저장 위치를 별도로 지정하지 않았다면 *C:\Logs*에 저장됩니다.

task-TSK-20190313-152013.log

```
Sim End.
Disposing ...2019-03-13 15:20:25
Elapsed Time = 00:00:00.0052814
Done ...2019-03-13 15:20:25



Mozart Function profiling
Name      Calls      Mean sec Total sec
SETUP_QUERY_ARGS_DEF   1          0.002     0.002
RUN_DEF 1          5.998     5.998
GET_WEIGHT_PRESETS0      1          0.003     0.003
RUN_DEF/GET_WEIGHT_PRESETS0     0          0          0
GET_EQP_LIST0     1          0          0
RUN_DEF/GET_EQP_LIST0    0          0          0
GET_EQP_SIM_TYPE_DEF     7          0          0
RUN_DEF/GET_EQP_SIM_TYPE_DEF    6          0          0
GET_DISPATCHER_TYPE_DEF          7          0          0
RUN_DEF/GET_DISPATCHER_TYPE_DEF          6          0          0
INITIALIZE_EQUIPMENT0       7          0.003     0.021
RUN_DEF/INITIALIZE_EQUIPMENT0       6          0          0
GET_WIPS0          1          0.005     0.005
RUN_DEF/GET_WIPS0          0          0          0
COMPARE_WIP_DEF          23         0          0
RUN_DEF/COMPARE_WIP_DEF          22         0          0
CREATE_LOAD_INFO0        24         0.002     0.048
RUN_DEF/CREATE_LOAD_INFO0        23         0          0
ON_BEGIN_LOCATE_WIP0     5          0          0
RUN_DEF/ON_BEGIN_LOCATE_WIP0     4          0          0
LOCATE_FOR_DISPATCH_DEF          5          0.007     0.035
RUN_DEF/LOCATE_FOR_DISPATCH_DEF          4          0          0
IS_BUCKET_PROCESSING0    19         0          0
```

- **Name**: Definition Name
- **Calls**: Definition 이 호출 된 수
- **Mean sec**: Definition 의 평균 실행 시간
- **Total sec**: Definition 의 총 실행 시간

MODEL

Model 개요 및 생성 방법

Model 의 개념

Model 은 MOZART 개발 Project 의 Logic 수행에 필요한 Input, Output Data (Schema) 및 실행시점에 Data 를 Logic 에서 사용할 수 있는 객체 형태로 전환하거나 Logic 의 수행결과를 Data File, Database 에 저장하기 위한 Query 와 설정정보를 저장하고 있는 정보파일입니다. 또한 Project 수행 방식이나 옵션을 위해 입력받을 수 있는 Argument 정보도 포함합니다. MOZART IDE 를 통해 개발되는 Project 의 경우 대부분 다양한 Input Data 를 사용하고 여러 Data Source 를 사용하게 되는 경우가 많으며 Model 을 통해 이러한 정보를 일관되게 관리하고 모델링할 수 있습니다.

Model 은 하나의 MOZART Project 에 복수 개를 생성할 수 있습니다. 최초 생성된 Model 의 내용을 상속받아 Child Model 을 생성하고 수정하여 사용할 수 있습니다. 다중 Model 생성 및 사용에 대해서 **Multi Model Project** 를 참조합니다

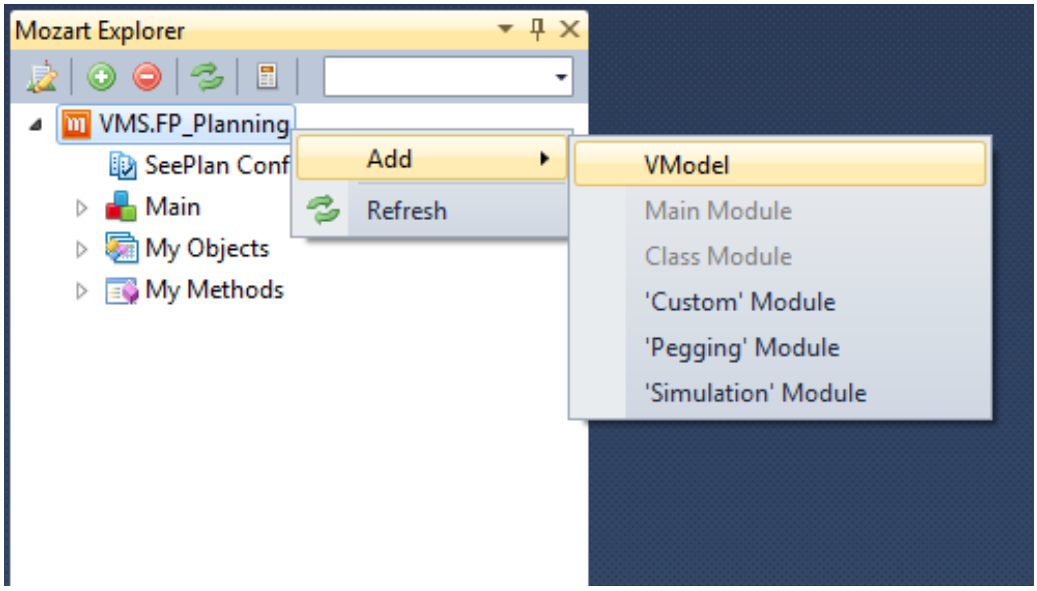
Model 의 구성요소

- Arguments : 실행 시 사용되는 Logic 실행의 옵션 및 설정값을 정의하는 기능입니다. Arguments 의 종류에는 Input Arguments 와 Config Arguments 가 있습니다.
- **Database** : Project 에서 사용되는 모든 Data 연결문자열을 관리하기 위한 기능입니다. 등록된 Data Source 는 Schema 편집 및 DataAction 편집 시 Data Source 속성에 설정값으로 사용이 가능합니다.
- Inputs : Task 의 Input Data Schema 와 DataAction 을 정의하고, 여러 Input Data 의 로드순서 및 방법을 정의하기 위한 기능입니다.
- **Outputs**: Task 의 Output Data Schema 와 저장순서, 방법을 정의하기 위한 기능입니다. 기본 Outputs 의 경우 파일로 저장되며, Outputs 정의 시 옵션 및 Query 지정을 통해 추가적으로 Database 저장이 가능합니다.

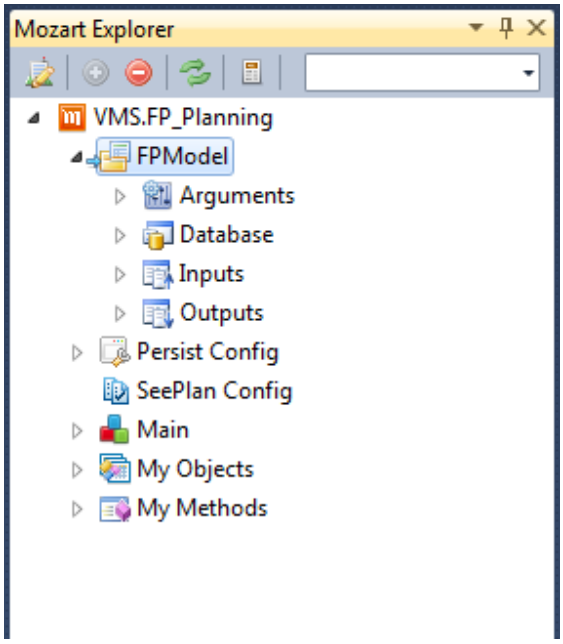
Project에 Model 추가하기

신규로 생성된 Project 에 Model 을 추가하는 방법은 아래와 같습니다.

1. MOZART Explorer 의 최상위 노드를 선택합니다.
2. 오른쪽 마우스 버튼으로 팝업메뉴를 활성화합니다.
3. **[Add > VModel]** 메뉴를 선택합니다.



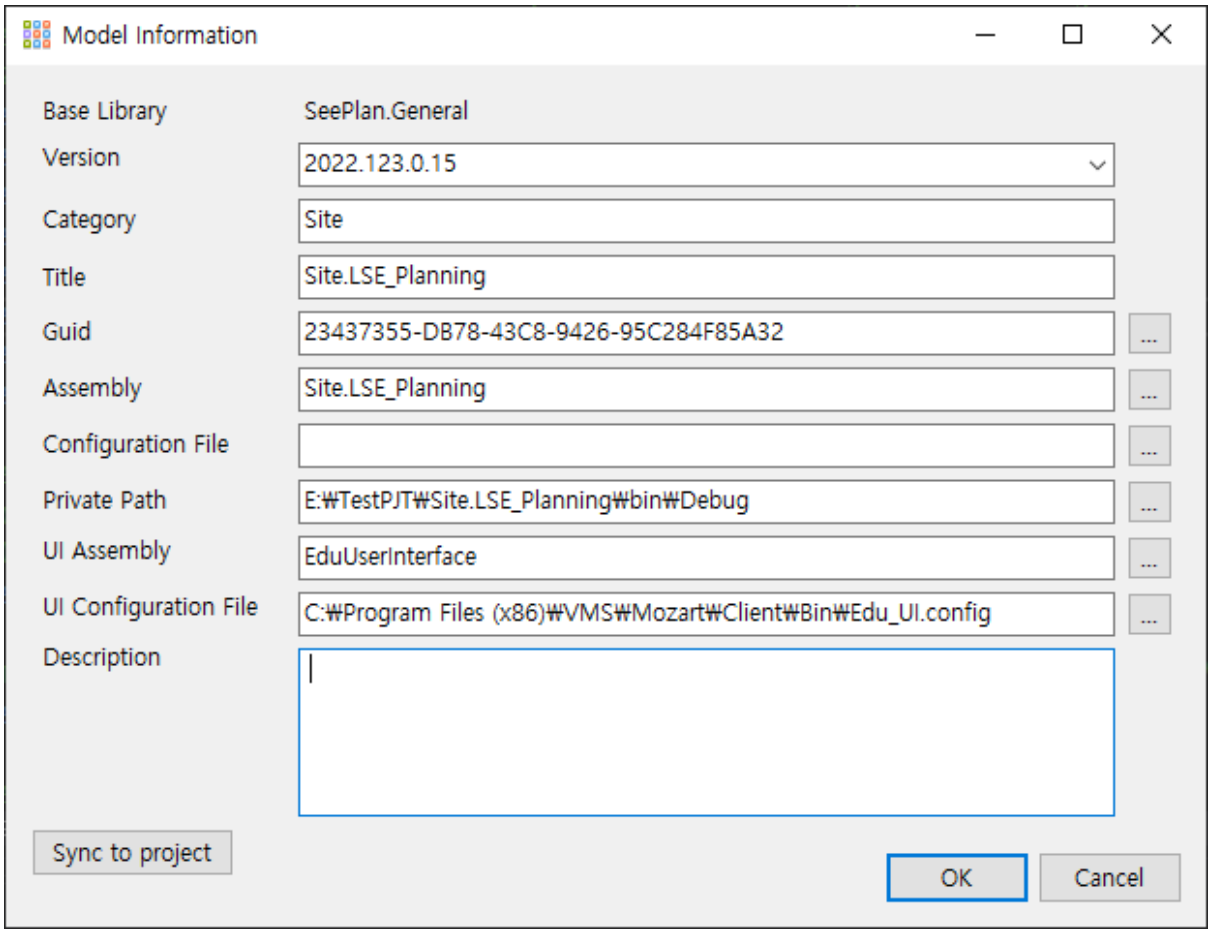
4. Model 명을 입력하는 다이얼로그가 활성화되면, 생성하려는 Model 명을 입력합니다.
5. MOZART Explorer 에서 입력된 Model 명의 Model 이 생성되었음을 확인합니다.



Model 정보 편집

Model 정보는 .vmodel 이라는 확장자로 저장이 되며 Model 의 구성요소에 대한 정보와 Model 의 속성정보를 포함합니다. MOZART IDE 를 통해 개발된 Task (Dll 파일) 는 반드시 Model 과 연계되어 수행할 수 있으므로 Model 에는 Model 을 실행할 dll 과 연관관계 정보, Model 실행을 위한 환경 설정 정보가 포함됩니다. 이 정보의 조회와 편집방법은 아래 절차를 따릅니다.

1. Model 노드를 선택하고 팝업메뉴로 Model 정보 편집메뉴 **[Edit Vinfo]** 를 실행하면 아래와 같은 Model 정보 편집창이 열립니다.



2. 필요한 항목에 대한 편집을 수행하고 저장합니다. 항목에 대한 설명은 아래와 같습니다.
   - Base Library : 소스 프로젝트와 연결된 도메인 라이브러리 이름입니다. ‘SeePlan.General’ 과 같이 SeePlan. 뒤에 표시되는 이름이 도메인 라이브러리입니다.

- Version : 소스 프로젝트와 연결되는 도메인 라이브러리의 버전을 선택해야 합니다. 선택 대상은 개발 PC에 설치된 Base Library의 버전을 선택할 수 있습니다.
- Category : Model 의 유형을 의미하며, 기본값은 Project 생성 시의 Site Prefix 로 설정됩니다.
- Title : Model 의 명칭입니다.
- Guid : Model 에 대한 GUID 입니다. Model 을 기반으로 만들어진 Task (Job Type) 를 구분하기 위한 ID 로 사용됩니다. 오른쪽 버튼을 사용해서 재생성할 수 있습니다.
- Assembly : Model 을 실행하기 위한 Dll 명입니다. 오른쪽 버튼을 사용해서 지정할 수 있습니다.
- Configuration File : Model 실행을 위한 설정파일을 지정합니다. 오른쪽 버튼을 사용해서 파일을 선택하거나 직접 입력할 수 있습니다.
- Private Path : 실행대상 Assembly 파일(Dll 파일)을 위치시키는 폴더명입니다. 오른쪽 버튼을 이용하여 사용자가 직접 지정할 수 있으며, 하단의 Sync to project 명령을 사용하면 Model 이 포함된 MOZART Project 의 “bin/Debug” 로 자동 설정이 됩니다.
- Description : Model 에 대한 설명 입니다.

3. 본 설정내용은 Model 파일(.vmodel) 에 저장되며 필요시 해당 정보만 파일로 Export 하는 것이 가능합니다. 팝업 메뉴의 [Export VInfo] 를 실행합니다.
4. 파일이 저장될 폴더를 지정하면 “Model 명.vinfo” 파일이 생성됩니다.
5. VInfo 파일에는 상기 Model 정보가 포함되며, 향후 MOZART Studio 에 사용자정의 User Interface 를 추가할 때 필요합니다. 생성된 파일의 내용은 아래 그림과 같습니다.

VInfo 파일 생성내용

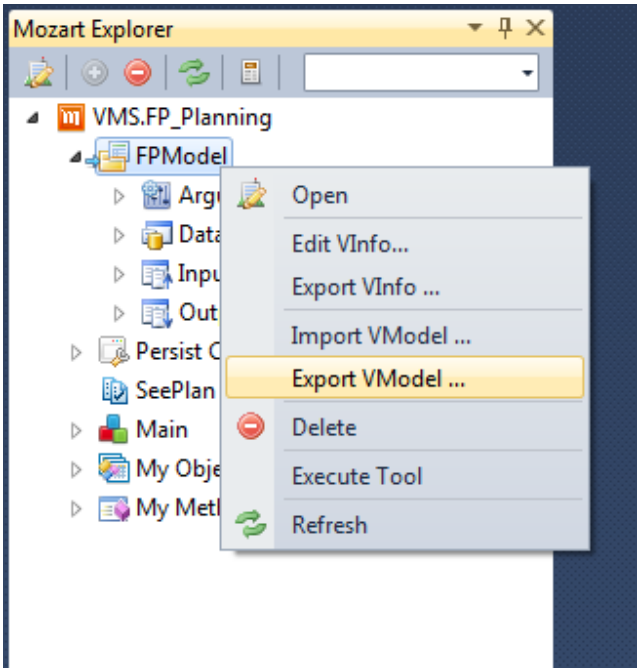
```
<target guid="7db113e2-1e7d-493a-8b2b-be22dd1b13ae" category="Site" title="VMS.FP_Planni
ng">
  <path configurationFile="" privatePath="D:\Mozart\VMS.FP_Planning\VMS.FP_Planning\bin
\Debug" />
  <type assemblyName="VMS.FP_Planning" typeName="" />
  <description />
</target>
```

Model Export, Import하기

유사한 Proejct 를 수행하게 될 경우 Model 을 복사하여 신규 Project 를 수행해야 하는 요구가 있을 수 있습니다. 이때에는 기존의 Source Model 을 Export 하여 신규 Model 에 Import 하여 사용할 수 있습니다.

Export 하기

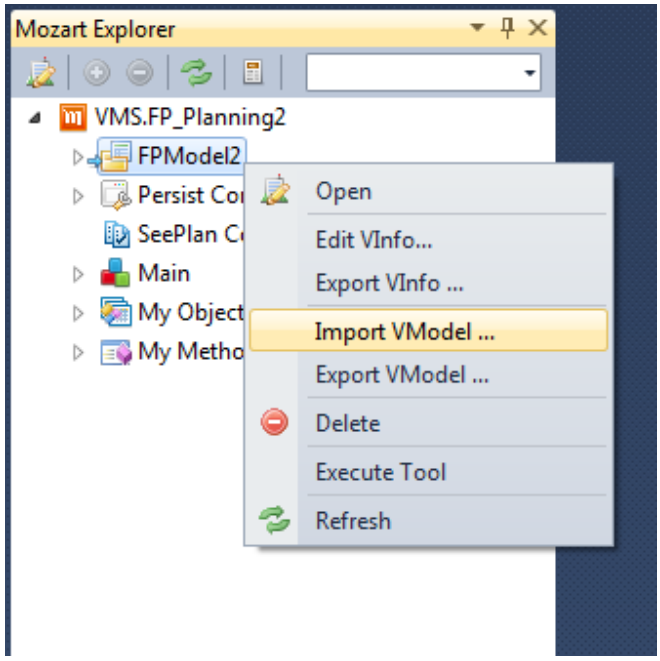
1. Model 노드에서 팝업메뉴의 [**Export VModel**] 메뉴를 실행합니다.



2. Model 저장 폴더를 지정한 후 저장합니다.

Import 하기

1. 신규 Project 에서 우선 빈 Model 을 추가합니다.
2. 추가된 Model 노드에서 팝업 메뉴의 [**Import VModel**] 메뉴를 실행합니다.



3. 기존에 Export 했던 .vmodel 파일을 선택합니다.
4. Model 정보를 확인해보면 신규 Project 에 맞도록 정보가 변경되었음을 확인할 수 있으며, 기존 Model 의 모든 Data 가 신규 Model 정보에 생성됨을 확인할 수 있습니다.

Arguments 관리

Arguments 는 실행 시 사용되는 Logic 실행의 옵션 및 설정값을 정의하는 기능입니다.

MOZART 의 Arguments 는 로직 구현시점에 사용하기 위해 정의할 수 있는 입력 파라미터입니다. 용도와 사용방식에 따라 다음 두 가지 방식으로 정의하여 사용할 수 있습니다.

- Input Arguments : 실행패널에서 조정할 수 있는 입력 파라미터입니다. 실행전에 실행조건 설정화면에서 사용자가 설정합니다.
- Config Arguments : 입력데이터와 바인딩하여 사용할 수 있는 입력 파라미터입니다. 실행시점에 입력데이터를 통해 파라미터 값이 설정됩니다.

Input Arguments

Task 실행 시 입력 Parameters 로 받아오는 Arguments 로 Module 실행 전체에서 사용되는 Context 와 상관없이 동일한 Parameters 값으로 사용되는 Arguments 를 정의합니다. 주로 Logic 의 실행 옵션에 대한 Parameters 로 사용합니다. 예를 들어 “장비 PM Schedule 반영 여부”, “수율적용여부” 와 같은 옵션 설정 시 주로 사용합니다. Input Arguments 의 경우 개발 시 MOZART Studio 의 Experiment 설정 시 설정하거나 Server 의 Job, Trigger 등록 시 Arguments 로 설정할 수 있습니다.

Input Arguments 의 경우 시스템에서 사전 정의된 Arguments 를 등록하여 Model 구동방식을 제어할 수 있습니다. (참조: **System Argument**)

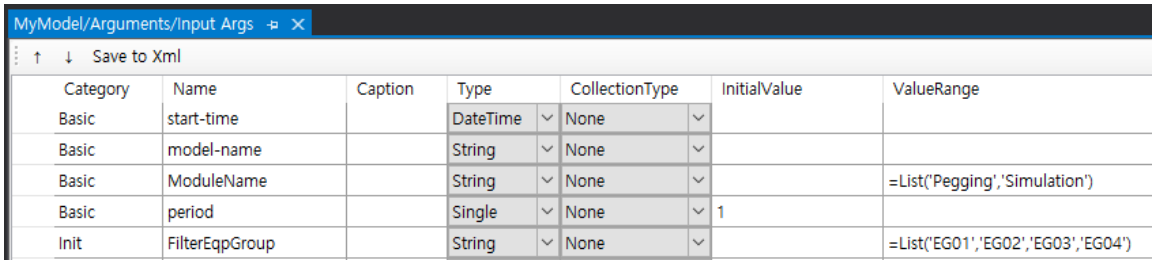
Arguments 기본 설정방법

Arguments 는 아래와 같은 절차로 설정합니다.

1. MOZART Explorer 의 등록된 Model 을 선택합니다.
2. 하위의 Arguments 노드를 펼치면 Input Args, Config Args 두 개의 노드가 있으며, 두 가지 중 등록이 필요한 Arguments 를 선택하여 더블 클릭합니다.
3. Arguments 를 등록할 수 있는 창이 열리면, 그리드의 Cell 별로 등록할 값을 입력합니다.
    - **Category** : Argument Group 으로 MOZART Studio 나 MMC 의 Job, Trigger 등록화면에서 Category 단위로 Argument 를 그룹핑하여 표시하게 됩니다.
    - **Name** : Argument 명
    - **Caption** : MOZART Studio 에서의 Argument 표시 명
    - **Type** : Argument Data 타입
    - **Collection Type** : Argument 를 집합변수로 생성할 수 있습니다. 사용가능한 형식은 List, Set, Dictionary 가 있으며, Dictionary 의 경우 Key 는 string 형식으로 정의됩니다.
    - **InitialValue** : Argument 의 기본 값
    - **ValueRange** : Argument 의 설정 값이 특정 범위의 값으로 제약될 때 해당 값의 범위를 입력합니다. 입력된 값의 Range 는 MOZART Studio 나 MMC 의 Parameter 입력 시, 콤보박스 형태로 표시하여 입력 받을 수 있습니다. Data 입력을 아래 그림과 같이 한 경우, MOZART Studio 의 [Experiment / Arguments] 입력창에서 해당 Argument 에 대해 리스트 에서 선택하여 입력할 수 있습니다. 체크 박스에서 여러 개의 값이 체크된 경우 하나의 string 값으로 들어옵니다.

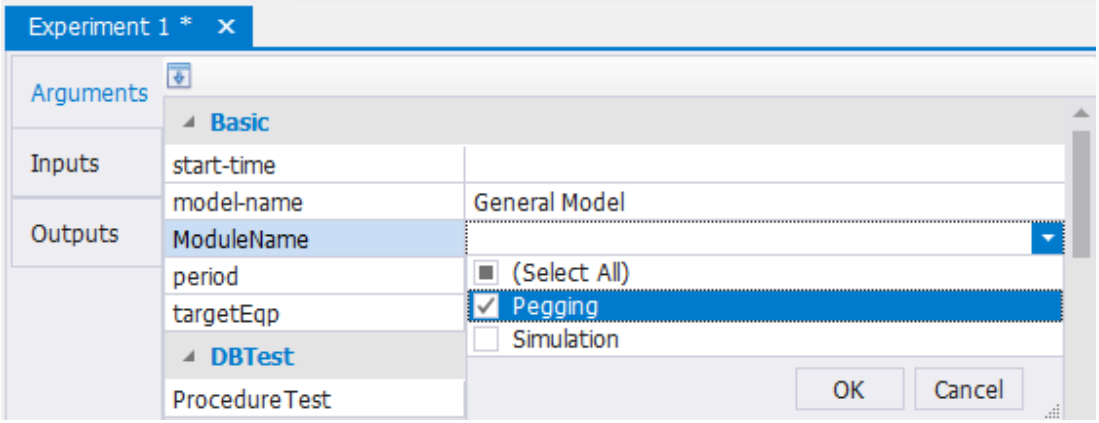
- **Description** : Argument 설명

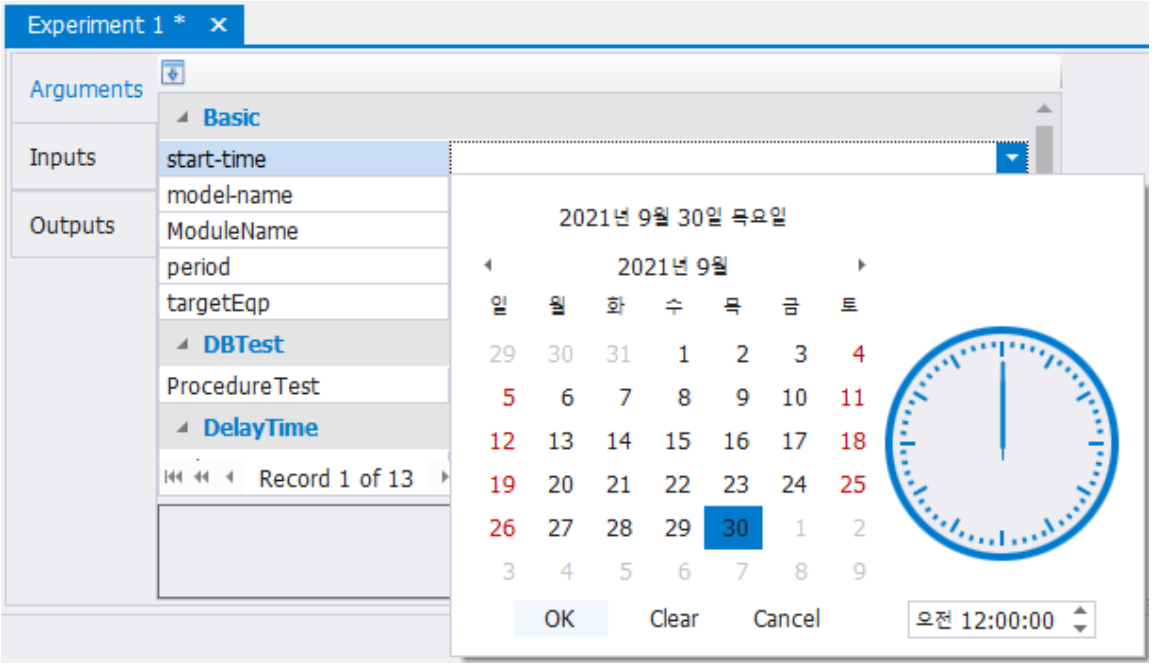
[입력 예]



Category	Name	Caption	Type	CollectionType	InitialValue	ValueRange
Basic	start-time		DateTime	None		
Basic	model-name		String	None		
Basic	ModuleName		String	None		=List('Pegging','Simulation')
Basic	period		Single	None	1	
Init	FilterEqpGroup		String	None		=List('EG01','EG02','EG03','EG04')

[Experiment / Arguments 입력화면 예]





4. 입력된 내용을 저장합니다. (**Ctrl + S** 혹은 상단 메뉴바의 [저장] 버튼 사용)
5. Argument 가 입력/수정되는 경우 MOZART Explorer 의 Arguments 왼쪽에 수정 여부가 표시됩니다.

6. MOZART Explorer 에 Focus 가 있는 상태에서 저장을 하면 Model 이 저장됩니다.
7. Model 이 정상적으로 저장되면 InputMart 를 통해 설정한 Argument 를 사용할 수 있습니다.

Input Arguments 참조방법

입력된 Arguments 의 값은 InputMart 를 통해 사용할 수 있습니다. Input Arguments 의 경우, InputMart 의 GlobalParameters 를 통해 참조할 수 있으며, 참조하는 방법은 아래의 예제 소스를 참고하시기 바랍니다.

Input Arguments 값 참조

```
// Input Arguments 로 SourceFolder, DestinationFolder 를 정의한 경우
// 아래와 같은 방식으로 참조 가능
string source = InputMart.Instance.GlobalParameters.SourceFolder;
string destination = InputMart.Instance.GlobalParameters.DestinationFolder;
```

Config Arguments

Input Arguments 와 유사하나 Config Arguments 는 Module 실행 시 Context 에 따라 달라지는 경우에 주로 사용합니다. 예를 들어 기본 공정이동 시간이나, 기본 Setup 시간 같은 경우 별도의 Input 에 의해 값이 처리될 수 있으나 Config 로 처리할 경우 라인별 공정 이동 시간을 달리하거나, 장비 그룹 별 Setup 시간을 달리 설정해야 하는 경우 Config Arguments 를 사용합니다. Config 의 값은 Input Data 를 만들고 이를 연결하여 사용하는 방식입니다. Config Arguments 설정방법에 대한 자세한 사항은 **Config Arguments 의 사용방법**을 참조하시기 바랍니다.

Config Arguments 참조방법

입력된 Arguments 의 값은 InputMart 를 통해 사용할 수 있습니다. Config Arguments 의 경우, InputMart 의 GetConfigParameters 를 통해 참조할 수 있으며, 참조하는 방법은 아래의 예제 소스를 참고하시기 바랍니다.

Config Arguments 값 참조

```
// Config Arguments 로 McGrpSetupTime 을 정의하고
// 해당 Config 값이 "EQPGRP1,60 ; EQPGRP2,45" 와 같을 때
ConfigParameters conf = InputMart.Instance.GetConfigParameters(string.Empty);
string[] grps = conf.mcGrpSetTimes.Split(';')
```

System Arguments 종류

MOZART Project 로 생성된 Task(Dll 파일) 와 Model 은 실제로 실행될 경우 MOZART 실행 엔진의 Model Task 를 통해 수행됩니다. 이러한 Model Task 의 실행 옵션을 조정하기 위해 사전 정의된 Arguments 가 System Arguments 입니다. 개발자는 Model 의 Input Arguments 설정 시 사전 정의된 System Arguments 를 지정함으로써 사전 정의된 실행 옵션을 활용할 수 있습니다. 아래 표는 사전 정의된 System Arguments 에 대한 설명합니다.

Basic Arguments

Argument Name	Argument 설명	Data Type
model	Model 파일 명(실행하는 위치의 경로가 포함되는 Full name)	string
experiment	Model 실행시 결과를 생성할 Experiment 명. 기본값은 “Experiment 1”	string
versionNo	Model Version 명 (기본값은 {model-name}-{yyyyMMdd-HHmmss} 형식)	string
model-name	versionNo 없을 때 versionNo에 사용될 이름	string
start-time	Task 시작시각 (Simulation Clock)	DateTime
end-time	Task 종료시각 (Simulation Clock)	DateTime
period	Plan&Schedule 기간	float
period-unit	period 설정 단위(기본은 일단위)	string
start-time.AdjustMinutes	Job 실행 시작 시간에서 해당한 시간만큼 조정한 시간을 start-time 으로 설정하도록 조정하는 입력변수	int
model-file	실행 Model 명을 포함한 Full Path	string
model-dll	Model 을 실행할 Dll 의 Full Path	string
model-config	Model 실행시 Config 파일의 Full Path	string

Data Download/Upload 옵션 Arguments

Argument Name	Argument 설명	Data Type
overwrite_result	Result 덮어쓰기 여부	bool
useDatabase	Database 읽기 여부	bool
saveDatabase	Output Data 를 Database 에 쓰기 여부	bool
db-To-file	시뮬레이션은 실행하지 않고 Database Sync 만 진행 여부 (기본값 = false)	bool
db-includes	Input Data Sync 대상 Table List 파일 이름, 대상 Table 들만 Sync 진행	string
db-excludes	Input Data Sync 제외 Table List 파일 이름, 제외 Table 들을 제외한 모든 Table Sync 진행,daction_includes 동일 값이 있는 경우, excludes 반영 안됨	string
daction_excludes	시뮬레이션 완료 후 DataAction 실행하지 않을 Table List, Comma 로 구분	string
daction_includes	시뮬레이션 완료 후 DataAction 실행할 Table List, Comma 로 구분	string
daction_excludes/in	시뮬레이션 시작 시 Input Schema 의 DataAction 중 실행에서 제외할 Action List, Comma 로 구분	string
daction_includes/in	시뮬레이션 시작 시 Input Schema 의 DataAction 중 실행할 Action List, Comma 로 구분	string

Run Arguments

Argument Name	Argument 설명	Data Type
more-runs	Model 의 반복 실행 횟수	int
more-config-[runindex]	반복 실행 회차 별 Arguments 설정 값이 저장된 xml 파일, config 파일에서 설정되지 않은 Arguments 는 이전 회차에서 사용한 Arguments 반영반복 실행 시 이전 회차에서 Loading 한 Data 들이 Memory 에서 지워지지 않아 다시 Loading 하면서 Error 가 발생할 수 있으므로 more-runs 사용 시 Result 기록 후 DataMart 를 Reset 해야 함	string

Zip Model Arguments

실행 후 압축파일로 Model 을 압축할 때 사용하는 규칙에 대한 설정 Arguments 입니다.

Zip Model Arguments

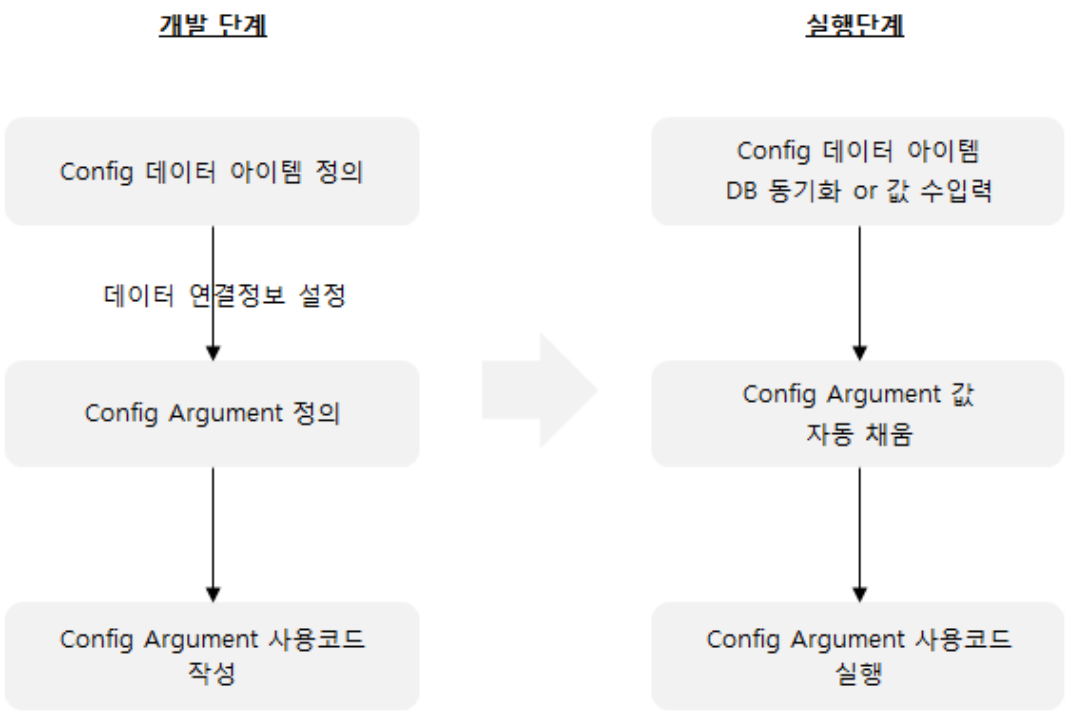
Aa Argument Name	Argument 설명	Data Type
zip	시뮬레이션 완료 후 Model 압축 여부, 우선 반영	bool
zip.FileNameTemplate	압축파일 이름 저장을 위한 Template 기본 Template 은 “${model_name}_${zip_now}${zip_postfix}”사용가능한 keyword 는 아래와 같습니다. * ${model_name} : Model 명* ${now} : 압축실행 시점 시각(DateTime)* ${zip_now} : 압축하는 시각문자열(format : yyyyMMddHHmmss)* ${zip_postfix} : 설정한 압축파일 이름 접미사* ${version-no} : Model 실행 Version 명	string

---

Config Arguments 설정방법

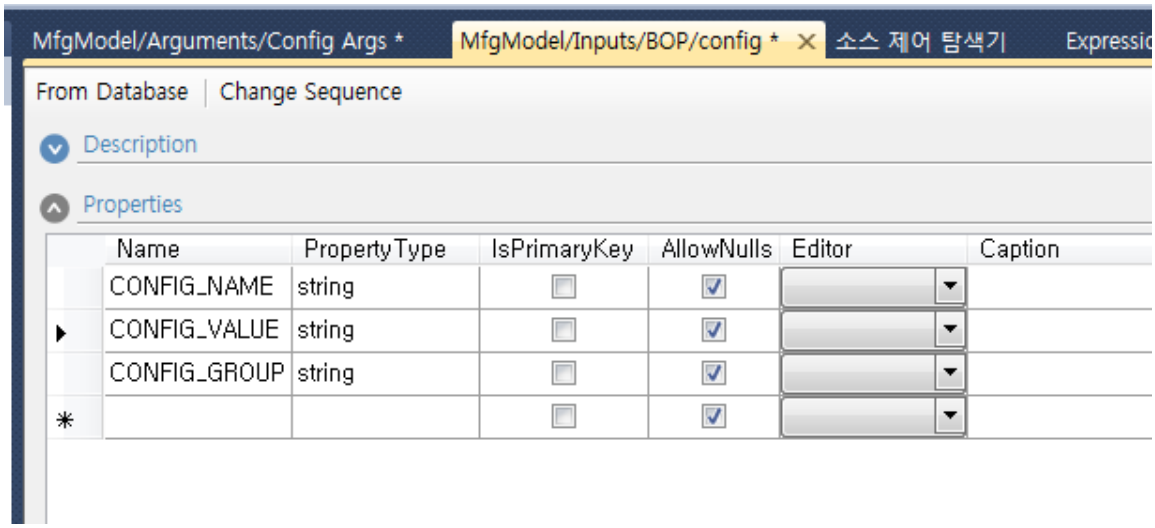
Config Argument 의 설정 방법을 설명합니다.

Config Arguments 의 Arguments 정의 방식은 Input Arguments 입력방식과 동일합니다. 단, Config 의 경우 사용자 조직에서 기존에 관리하던 기준정보를 참조해야 하는 경우가 많으므로, 이를 위해 Input DataItem 으로 실제 Arguments 값을 관리하고, 실행시점에 해당 DataItem 값을 자동으로 Config Arguments 에 할당하는 방식을 사용할 수 있도록 기능이 구성되어 있습니다. 아래 그림과 같습니다.

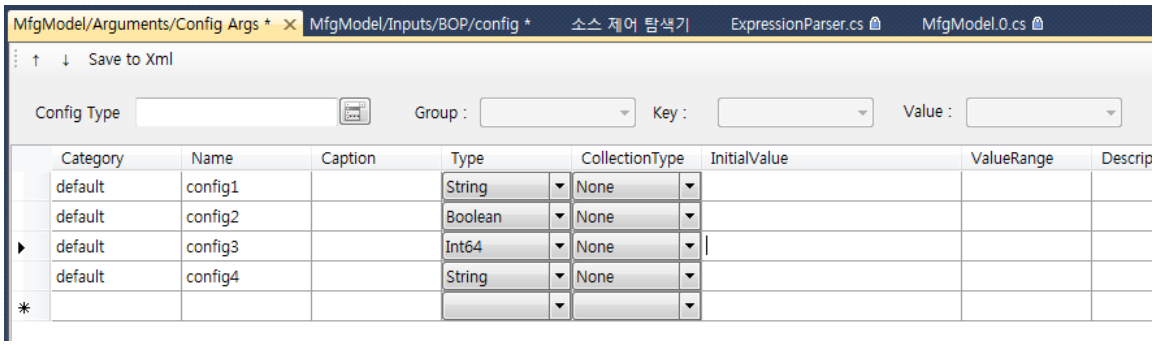


개발단계에서의 Config Arguments 정의&사용방식

1. Config Data 를 저장할 DataItem 을 Inputs 에 생성합니다. (아래 그림)



2. 시스템에서 사용하게 될 Config Arguments 를 정의합니다. (아래 그림)

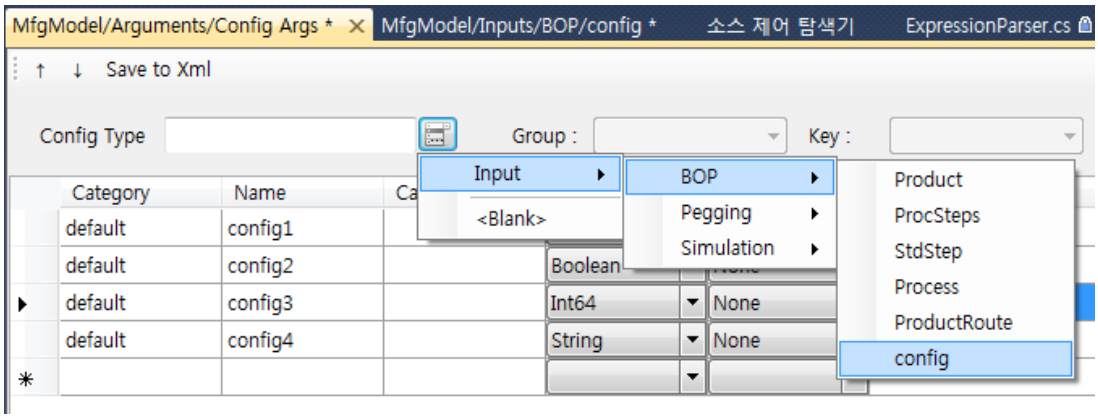


3. Config Arguments 를 자동으로 할당할 때 사용할 DataItem 정보 및 Column 매핑 정보를 설정합니다.

   실행시점에 Config Arguments 의 값을 채울 수 있는 Config Data Table 값과 Mapping Field 에 대한 설정을 추가적으로 수행합니다.

   - **Config Type** : Config Arguments 의 Data Table 을 Input DataItem 중 선택하여 입력합니다.
   - **Group** : Input DataItem Data 를 Grouping 하여 관리하고자 할 때 사용하며, DataItem 의 값을 설정할 때 해당 Column 에 Group 을 설정합니다. 선택한 Config Type 의 Column 중 선택할 수 있으며 필수항목은 아닙니다.

- **Key** : Config Arguments 의 Name 속성과 Mapping 할 Column 을 지정합니다. 선택한 Config Type 의 Column 중 선택할 수 있습니다.
- **Value** : Config Arguments 의 값을 Mapping 할 Column 을 지정합니다. 선택한 Config Type 의 Column 중 선택할 수 있습니다.



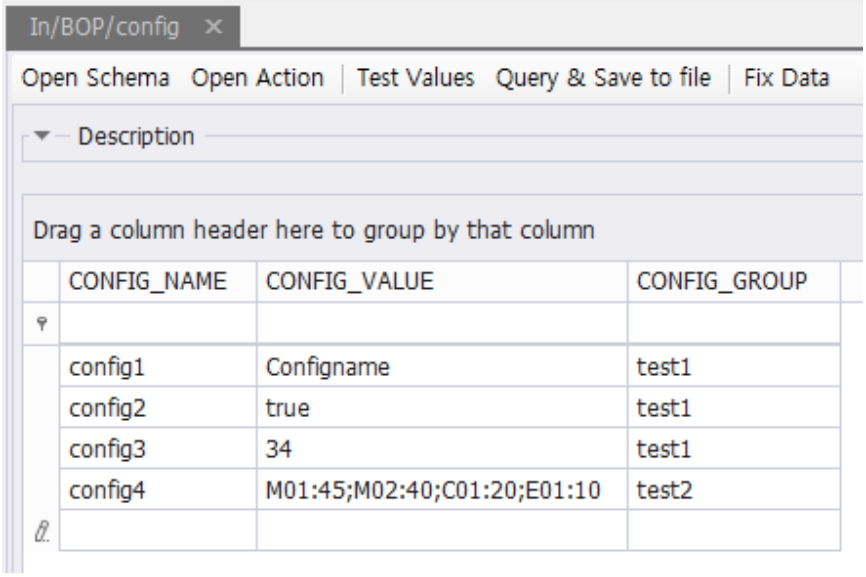
4. 설정된 Config Arguments 를 사용하여 Logic 을 구현합니다. 아래는 예시적인 코드를 보여줍니다.

Config Arguments 사용 예

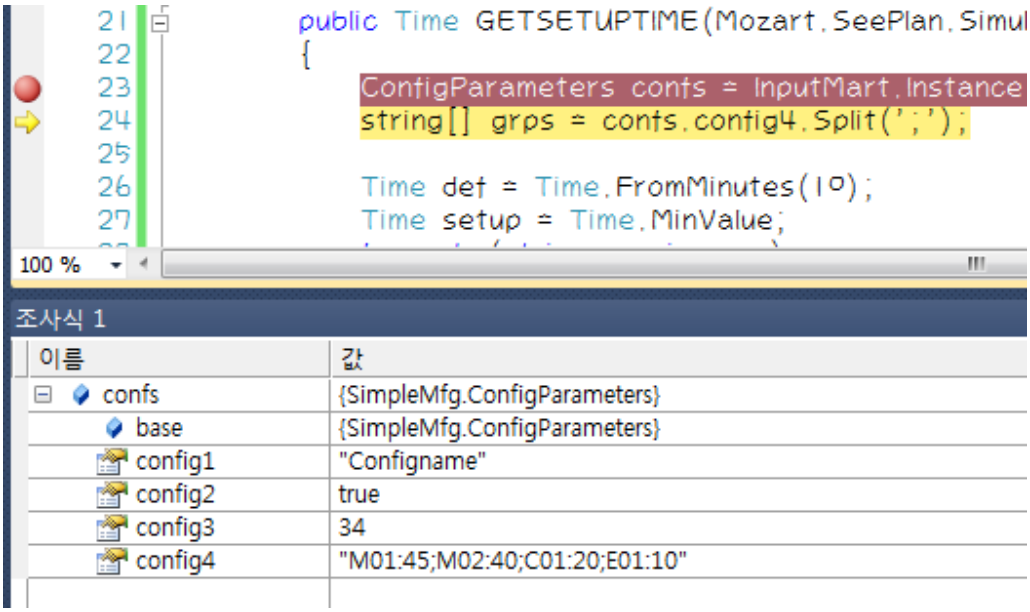
```
// Setup 시간을 확인할때 장비 Group 에 따라 다르게 적용하기 위한 코딩 예
 // config4 를 장비별 Setup 시간을 정의한 Config Arguments 로 설정했을 때
public Time GETSETUPTIME(Mozart.SeePlan.Simulation.AoEquipment aeqp, IHandlingBatch hb,
 ref bool handled, Time prevReturnValue)
{
   ConfigParameters confs = InputMart.Instance.GetConfigParameters(string.Empty);
   string[] grps = confs.config4.Split(';');

   Time def = Time.FromMinutes(10);
   Time setup = Time.MinValue;
   foreach (string grp in grps)
   {
       string[] setupTime = grp.Split(':');
       if (aeqp.EqpID == setupTime[0])
       {
           setup = Time.FromMinutes(Convert.ToInt32(setupTime[1]));
           break;
       }
   }

   if (setup == Time.MinValue)
   setup = def;

   return setup;
}
```

실행단계에서 확인

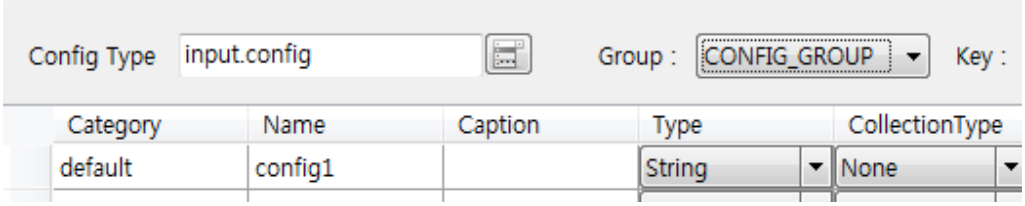
1. MOZART Studio 에서 Config 정보로 사용할 Data 를 확인합니다. 만일 DB 로부터 Data 를 Sync 하도록 DataAction 을 추가한 상태라면 [Query & Save to file] 메뉴를 사용하여 값을 가져옵니다.



2. 위와 같이 설정한 경우 Config Parameters 의 Config 명 변수에 값이 자동으로 할당되고 상기 로직 구현 예제와 같이 직접 참조가 가능하게 됩니다. 아래 그림은 Debugging 시점에 해당 confs 변수의 값 할당 현황을 보여줍니다.



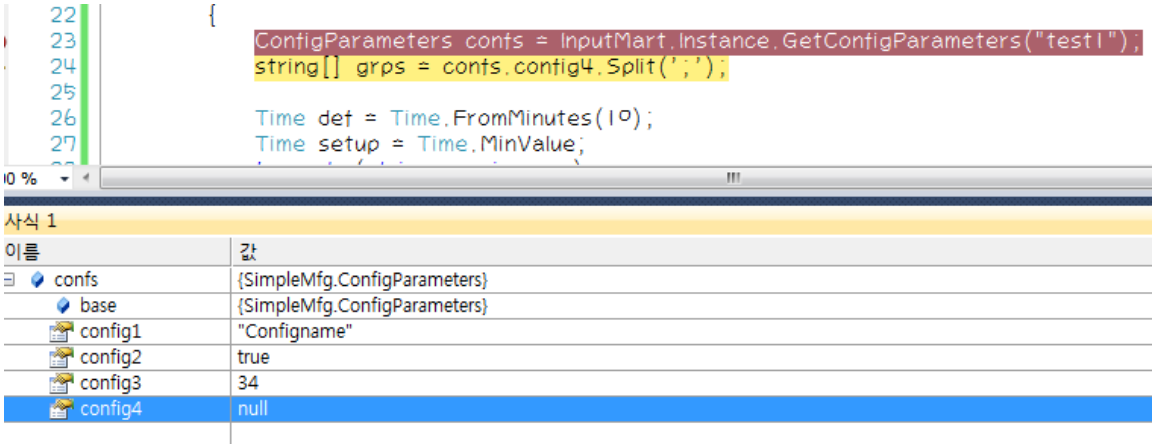
3. 라인별로 혹은 특정 Group 에 대한 Config 를 별도로 관리하고자 할 경우에는 Config Arguments 설정 시 Group Mapping Column을 선택한 후 아래와 같은 방식으로 Config 변수를 가져올 때 특정 Group 을 지정하는 방식을 사용할 수 있습니다.



Group별 Config Arguments 받아오기

```
// 전체를 받아오는 경우
 ConfigParameters confs = InputMart.Instance.GetConfigParameters(string.Empty);
 // 특정 Group 을 받아오는 경우, 아래와 같이 설정하면 config Table 의 CONFIG_GROUP = "test1"
 인 세 개의 설정정보만 Mapping 값이 설정되어 생성됩니다.
 ConfigParameters confs = InputMart.Instance.GetConfigParameters("test1");
```

4. 위와 같이 특정 Group 에 대한 설정 시 결과 값은 아래와 같이 변경되어 “config4”에 대한 값이 null 로 setting 됩니다.



Data Source 관리

MOZART Project 에서 사용할 수 있는 Data Source 를 등록하고 관리하기 위한 기능입니다. 한 번 등록된 Data Source 는 Inputs/Outputs Data 를 정의하고 DataAction 을 생성하기 위

한 Data Source 로 사용됩니다. MOZART 의 Data Source 는 복수 개의 연결문자열 정보를 포함할 수 있도록 구성되며, 첫 번째 Data 연결문자열이 동작하는 방식을 사용합니다. 필요 시 코드 내부에서 Data Source 의 연결문자열 명으로 Data Source 를 지정할 수 있습니다.

Data Source 관리방법

1. MOZART Explorer 의 Database 노드를 선택합니다.
2. Database 노드에서 오른쪽 마우스 버튼을 클릭하여 **[New DataSource…]** 팝업메뉴를 실행합니다.
3. Edit Data Source 다이얼로그에서 Data Source 명을 입력한 후, **[Add]** 버튼을 사용하여 연결문자열을 등록합니다.
4. Data Source 다이얼로그에서 Data 연결문자열 명을 입력한 후, Data Provider 콤보박스에서 대상이 되는 Data Provider 를 선택합니다.

   > 💡 **Note**
   > 지원대상이 되는 Provider 는 Microsoft Access, Microsoft SQL Server, Oracle, DB2, SQLite, MySql, ProgreSQL 일곱 가지입니다.

5. Data 연결문자열을 입력합니다. 문자열은 하단의 텍스트박스에 직접 입력하거나, […] 버튼을 이용하여 Data Source 별 입력창을 통해 입력할 수 있습니다.
   - **[…]** 버튼을 클릭하면 Select database 다이얼로그가 열립니다. “**Select a database**” 콤보박스에서 기본에 등록된 Database 를 선택할 수 있으며, 신규로 추가할 경우에는 <**New Database Connection**>을 선택합니다.
   - 신규로 추가하는 경우 하단의 Select a database type 콤보박스에서 database type 을 선택한 후 **[Create…]** 버튼을 클릭합니다.
   - Database Type 에 따른 입력창을 통해 접속정보를 입력하면 Data Source 가 추가됩니다.
6. 연결문자열이 등록되면 Data Source 다이얼로그 하단 리스트에 등록된 모든 연결문자열이 표시됩니다.
7. 연결문자열을 삭제할 때에는 삭제 대상인 문자열을 리스트에서 선택하고 **[Remove]** 버튼을 사용하여 삭제할 수 있습니다.
8. 연결문자열 편집 시에는 해당 문자열을 리스트에서 더블클릭하거나 선택 후 **[Edit]** 버튼을 사용합니다.

9. Data Source 에 여러 문자열이 있는 경우, 특별한 설정을 하지 않으면 첫 번째 문자열이 적용됩니다. 적용되는 문자열을 변경하고자 하는 경우 **[Up][Dn]** 버튼을 사용합니다.

Data Item 관리

MOZART 는 제조업의 Planning, Scheduling 시스템 개발을 위한 도구로서 다량의 복잡한 Data 를 가공하고 처리하는 Logic 개발을 위주로 합니다. MOZART 의 Inputs Modeling 도구는 이러한 Logic 개발을 위한 Input Data 를 쉽게 가공할 수 있도록 지원합니다. MOZART Inputs 는 DataItem 과 이를 Loading 하기 위한 Persist Configuration 으로 구성됩니다. DataItem 은 Data Schema 와 DataAction 으로 정의되며, 이 중 한 가지만 존재할 수도 있습니다. 또한 MOZART Project 를 통해 개발된 Logic 의 결과로 다양한 Output 이 생성될 수 있습니다. MOZART 의 Output Modeling 도구는 이러한 다양한 Output 을 손쉽게 정의하고 참조할 수 있도록 지원하며, File 과 Database 로 저장하는 일련의 과정을 쉽게 처리할 수 있는 기능을 제공합니다. Output 을 기록하기 위해서는 Output Data 를 정의하고 Model 의 실행 시에 해당 Data 에 값을 기록한 후, 최종적으로 File 및 Database 에 저장하는 세 단계를 거칩니다. Output Data 는 Input DataItem 을 정의하는 것과 동일한 방식으로 정의할 수 있습니다.

DataItem 생성/편집

정의된 Input DataItem 은 RunTime 시에 Memory 에 로딩되어 Logic 개발 시 Input Data 로 활용됩니다. 이때, 정의된 Schema 를 기반으로 한 DAO(Data Access Object) 형태로 바로 사용할 수도 있고, Data 로딩 시점에 해당 Data 를 가공하여 사전에 정의된 DMO(Data Manipulation Object) 형태로 저장하여 사용할 수도 있습니다. 반면 Output DataItem 의 경우, 저장 시 자동으로 DAO 클래스가 생성되며 Project 의 OutputMart 에 EntityTable 형태로 등록되어 Logic 개발 시점에 쉽게 코딩 한 줄 없이 Output Data 를 참조하는 것이 가능합니다. 실행 코드에서 이렇게 참조한 Output DataItem 에 결과를 업데이트 하면 Memory 상에 원하는 Output 이 생깁니다. MOZART IDE 에서 Inputs 는 **[Persist Config / Input Config]** 메뉴를 통해 Input DataItem 에 대해 Data 로딩 순서, 로딩 방식 및 로딩 처리를 위한 함수를 정의하고 조정하는 작업을 수행할 수 있으며, Outputs 는 **[Persist Config / Output Config]** 메뉴를 통해 Memory 상의 Output Data 를 최종적으로 File, DB 와 같은 최종 Data 저장 장치에 기록하는 일련의 과정을 설정을 통해 수행할 수 있습니다.

Input Persist Configuration 정의

[Output Persist Configuration 정의]

DataItem 생성/편집 방법

DataItem 은 Data Schema 에 Data 를 채우거나 Schema의 Data 를 외부 Database 에 저장하기 위한 Query 집합인 DataAction 으로 구성됩니다. MOZART 에서는 Query 관리를 위해 DataAction 이라는 개념을 사용합니다.

Inputs, Outputs 모두 동일한 구조의 DataItem 을 사용합니다. Schema 와 DataAction 은 1:N 의 관계를 가집니다. 즉, 하나의 DataItem 은 한 개의 Schema 와 1개 이상의 DataAction 으로 구성될 수 있습니다. Outputs 의 경우 Database 에 저장이 필요없을 때에는 DataAction 없이 Schema 만 생성할 수 있습니다. DataAction 또한 하나의 Query 만으로 구성되지 않으며, 내부적으로 복수 개의 Query(Command) 를 포함하며 실행할 때는 등록된 Command 순서에 따라 순차적으로 실행합니다. 따라서 특정Table 에 Data 를 입력하기 전에 해당 Table 의 Data 를 지워야 하는 경우, 하나의 DataAction 에 Delete Command 와 Insert Command 를 순차적으로 실행할 수 있도록 구성되어 있습니다.

[Schema 편집]

[DataAction 편집]

DataItem 정보 Import/Export 하는 방법

Mozart IDE 또는 Mozart Studio에서는 DataItem 정보를 Import 또는 Export 할 수 있습니다. 스키마가 추가되거나 변경되었을 때 해당 모델을 운영에 적용하기 전 현재 운영 중인 모델에 스키마를 추가하여 테스트할 때 사용하거나, 신규 프로젝트를 진행할 때 비슷한 스키마가 다른 프로젝트에 있는 경우 새로 정의하지 않고 가져오는 용도로 사용이 가능합니다.

DataItem Import/Export 주요 사항

파일 확장자

Mozart IDE 또는 Mozart Studio에서 Export된 DataItem의 확장자는 다음과 같습니다.

- 개별파일: vditem

- Input 또는 Output 파일 집합: vditems

> **Note**
> Mozart IDE와 Mozart Studio의 DataItem 정보 파일의 확장자가 통일된 것은 2021.119.2 버전 부터입니다. 이전 버전에서 Mozart IDE에서 Export한 vditem을 Mozart Studio에서 Import 하려면 Export 후 파일 확장자를 vdif로 변경하시면 됩니다.

메뉴 설명

Mozart IDE에서 DataItem을 선택하여 마우스 우 클릭으로 Context Menu를 열면 Import, Export 메뉴를 사용할 수 있습니다. Import, Export 단위에 따라 사용되는 명령은 다음과 같습니다.

- **Import, Export > [DataItem(N)...]**: 선택한 DataItem 1개 또는 복수의 DataITem을 Import 또는 Export할 수 있습니다. (vditem)
- **Import, Export > [DataItems...]**: 선택한 DataGroup 이 속한 Input 또는 Output 전체를 Import, Export합니다. (vditems)

Import/Export되는 정보들

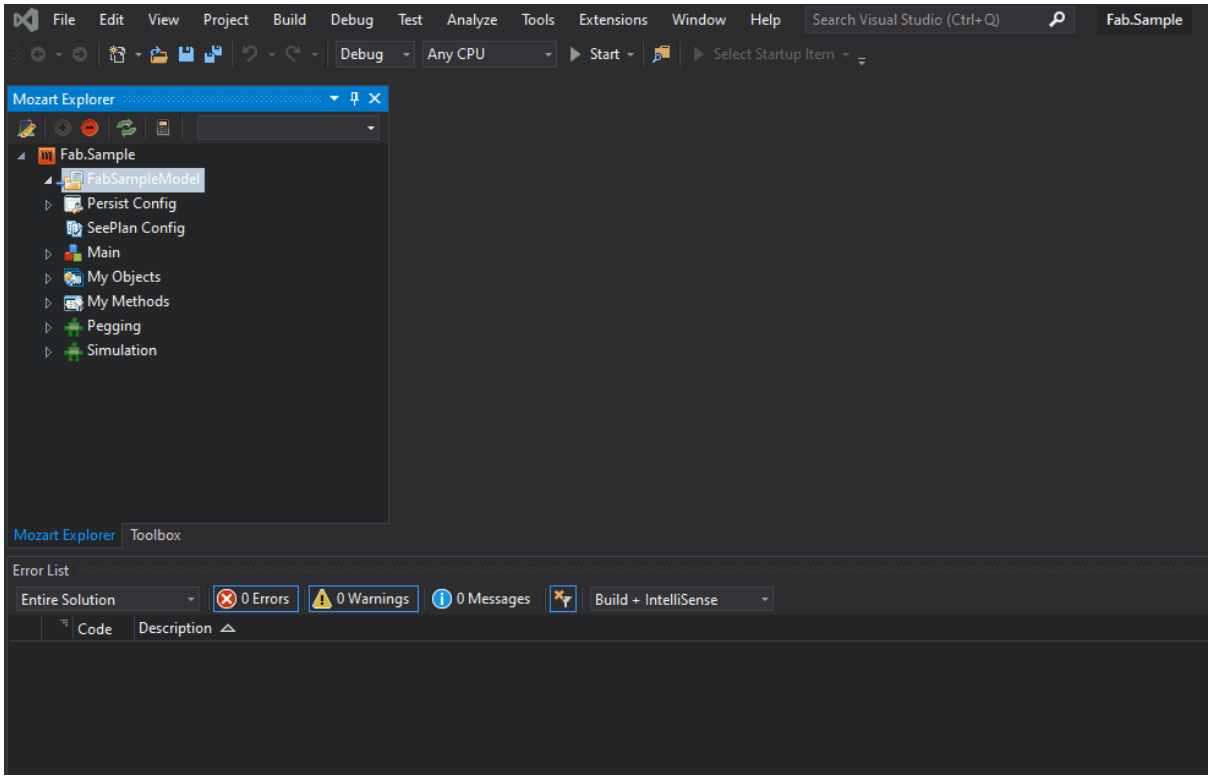
DataItem을 Import, Export하면 다음 정보들이 포함됩니다.

- DataItem 스키마 정보
- DataAction 구현 및 설정 내용

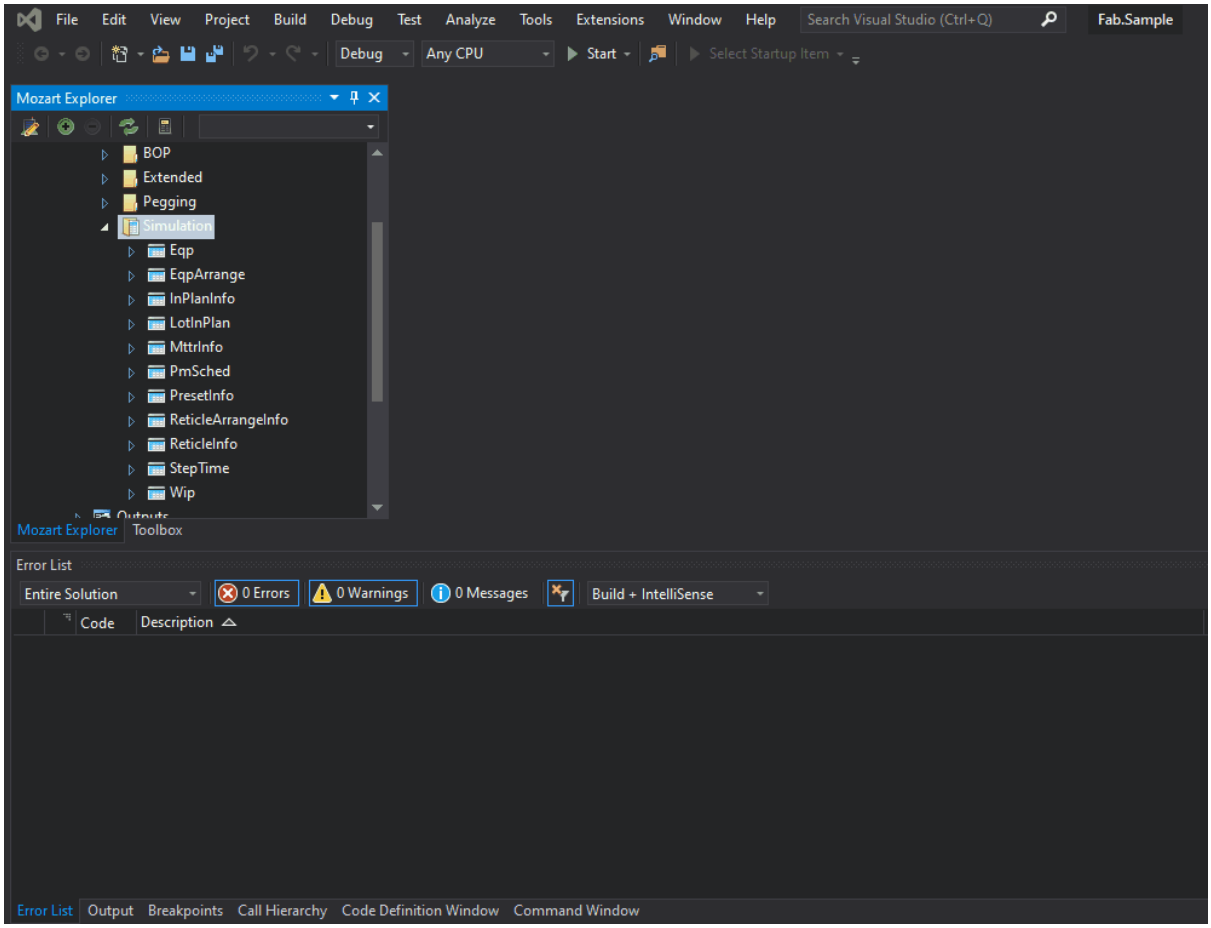
Connection String이 포함된 DataSource의 경우 DataSource를 별도로 Import/Export 해야 합니다.

DataItem Export 예시

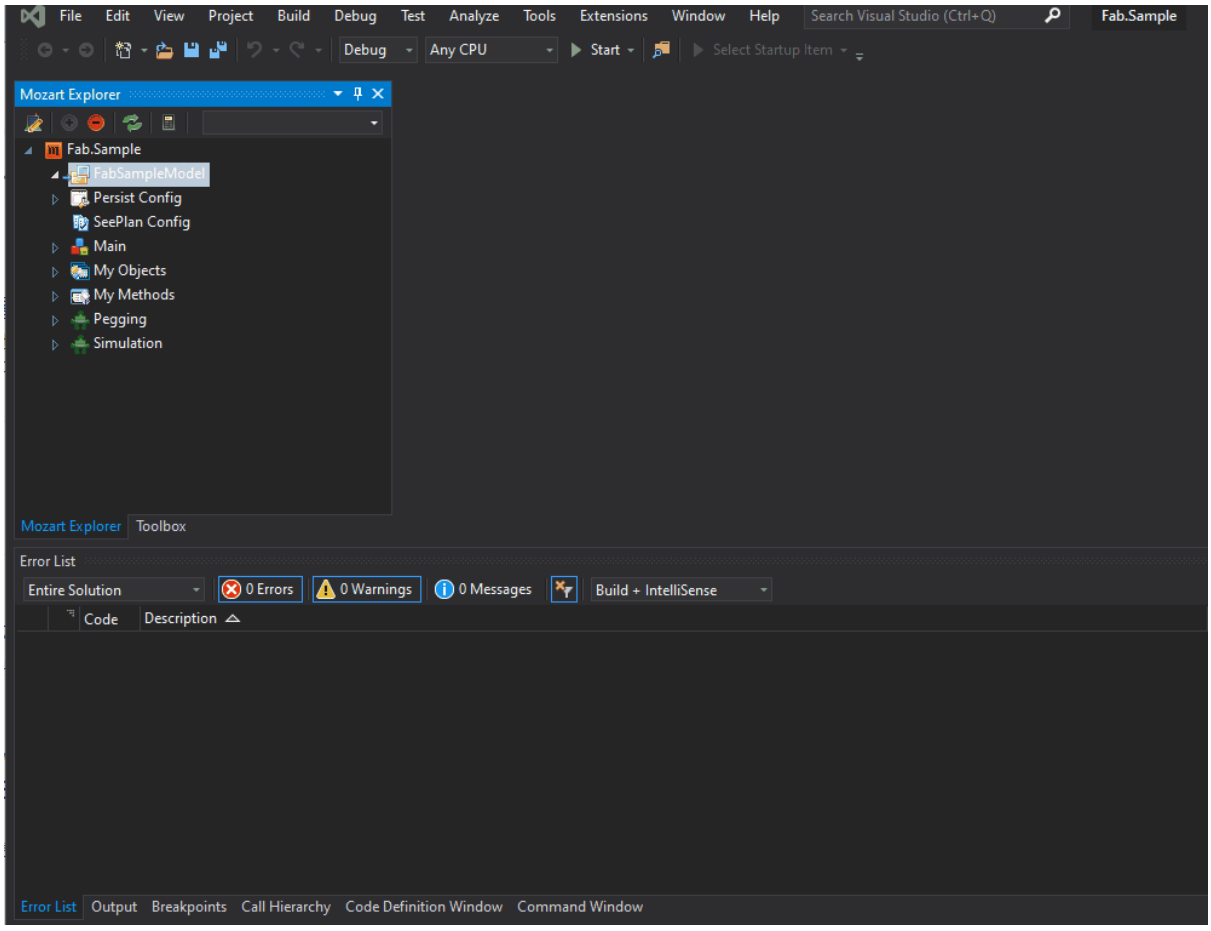
단일 DataItem Export하기



다중 DataItem Export하기

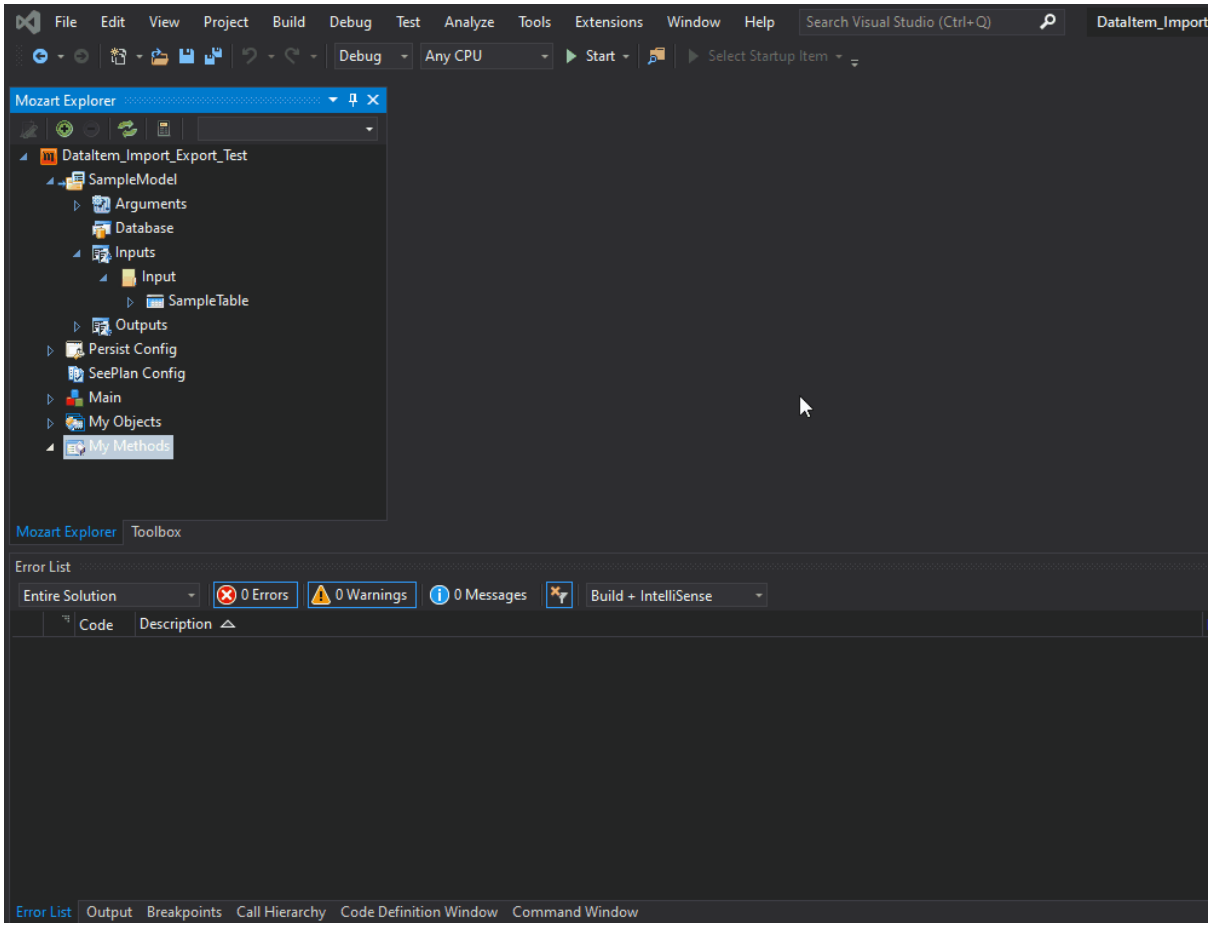


스키마 전체 Export하기

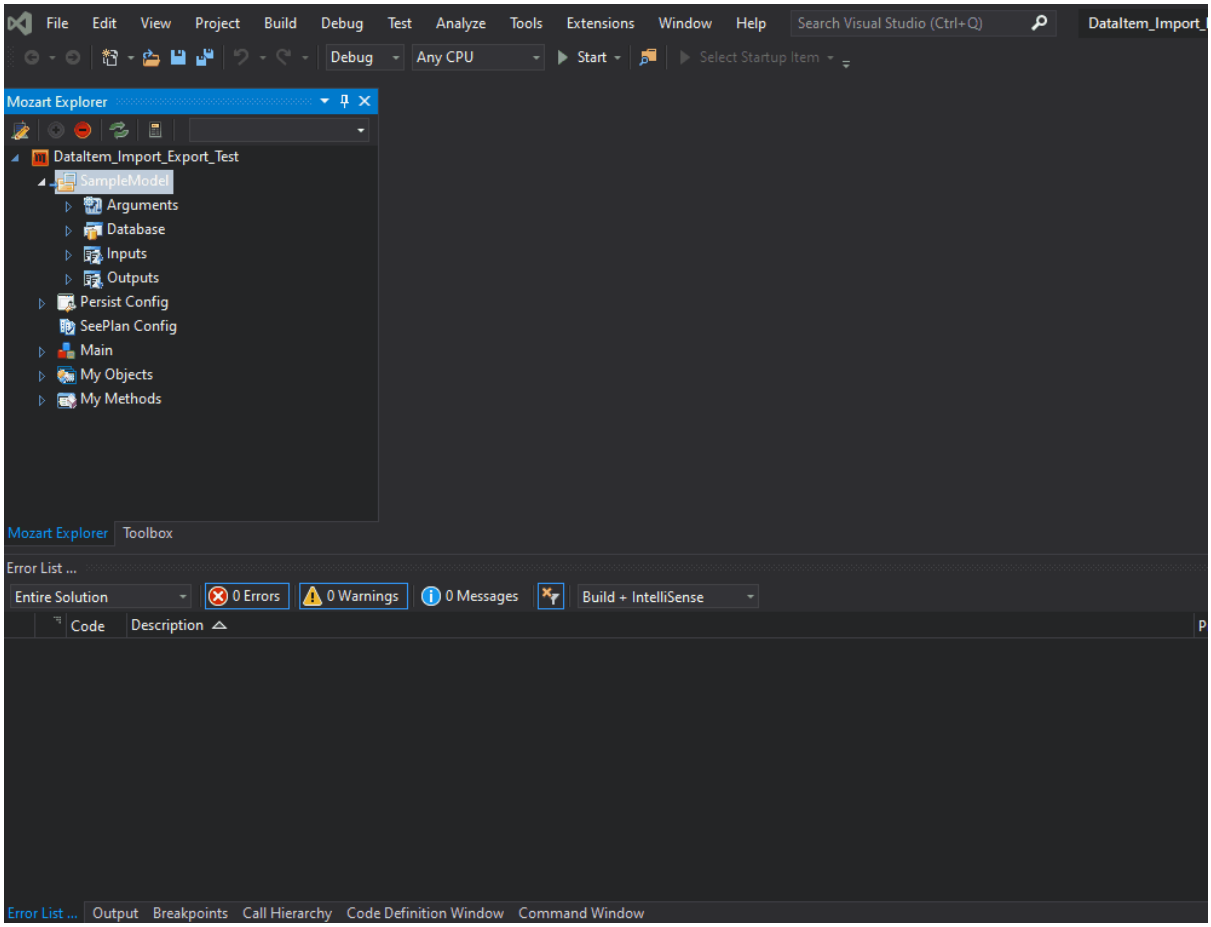


DataItem Import 예시

단일/다중 DataItem Import하기



스키마 전체 Import하기



Mozart IDE에서 Export한 DataItem을 Mozart Studio에서 Import하려면 [DataItem 정보 Import/Export 하는 방법](#)을 참조 바랍니다.

DataItem 정보 Import/Export 하는 방법(Studio)

Mozart IDE 또는 Mozart Studio에서는 DataItem 정보를 Import 또는 Export 할 수 있습니다. 스키마가 추가되거나 변경되었을 때 해당 모델을 운영에 적용하기 전 현재 운영 중인 모델에 스키마를 추가하여 테스트할 때 사용하거나, 신규 프로젝트를 진행할 때 비슷한 스키마가 다른 프로젝트에 있는 경우 새로 정의하지 않고 가져오는 용도로사용이 가능합니다.

DataItem Import/Export 주요 사항

파일 확장자

Mozart IDE 또는 Mozart Studio에서 Export된 DataItem의 확장자는 다음과 같습니다.

- 개별파일: vditem
- Input 또는 Output 파일 집합: vditems

> 💡 **Note**
> Mozart IDE와 Mozart Studio의 DataItem 정보 파일의 확장자가 통일된 것은 2021.119.1 버전 부터입니다. 이전 버전에서 Mozart IDE에서 Export한 vditem을 Mozart Studio에서 Import 하려면 Export 후 파일 확장자를 vdinfo로 변경하시면 됩니다.

메뉴 설명

Mozart IDE에서 DataItem을 선택하여 마우스 우 클릭으로 Context Menu를 열면 Import, Export 메뉴를 사용할 수 있습니다. Import, Export 단위에 따라 사용되는 명령은 다음과 같습니다.

- **Import, Export > [DataItem(N)...]**: 선택한 DataItem 1개 또는 복수의 DataITem을 Import 또는 Export할 수 있습니다. (vditem)
- **Import, Export > [DataItems...]**: 선택한 DataGroup이 속한 Input 또는 Output 전체를 Import, Export합니다. (vditems)

Import/Export되는 정보들

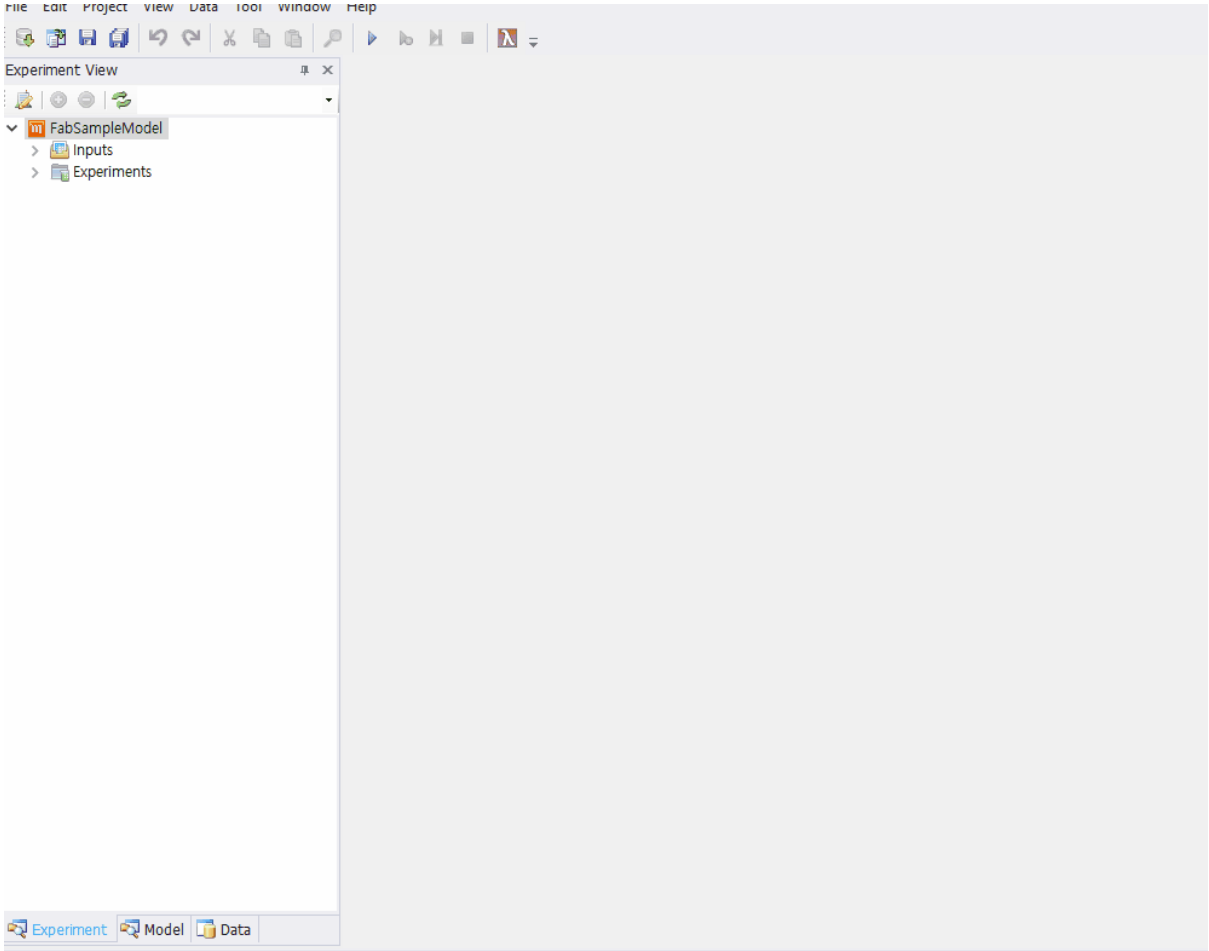
DataItem을 Import, Export하면 다음 정보들이 포함됩니다.

- DataItem 스키마 정보
- DataAction 구현 및 설정 내용

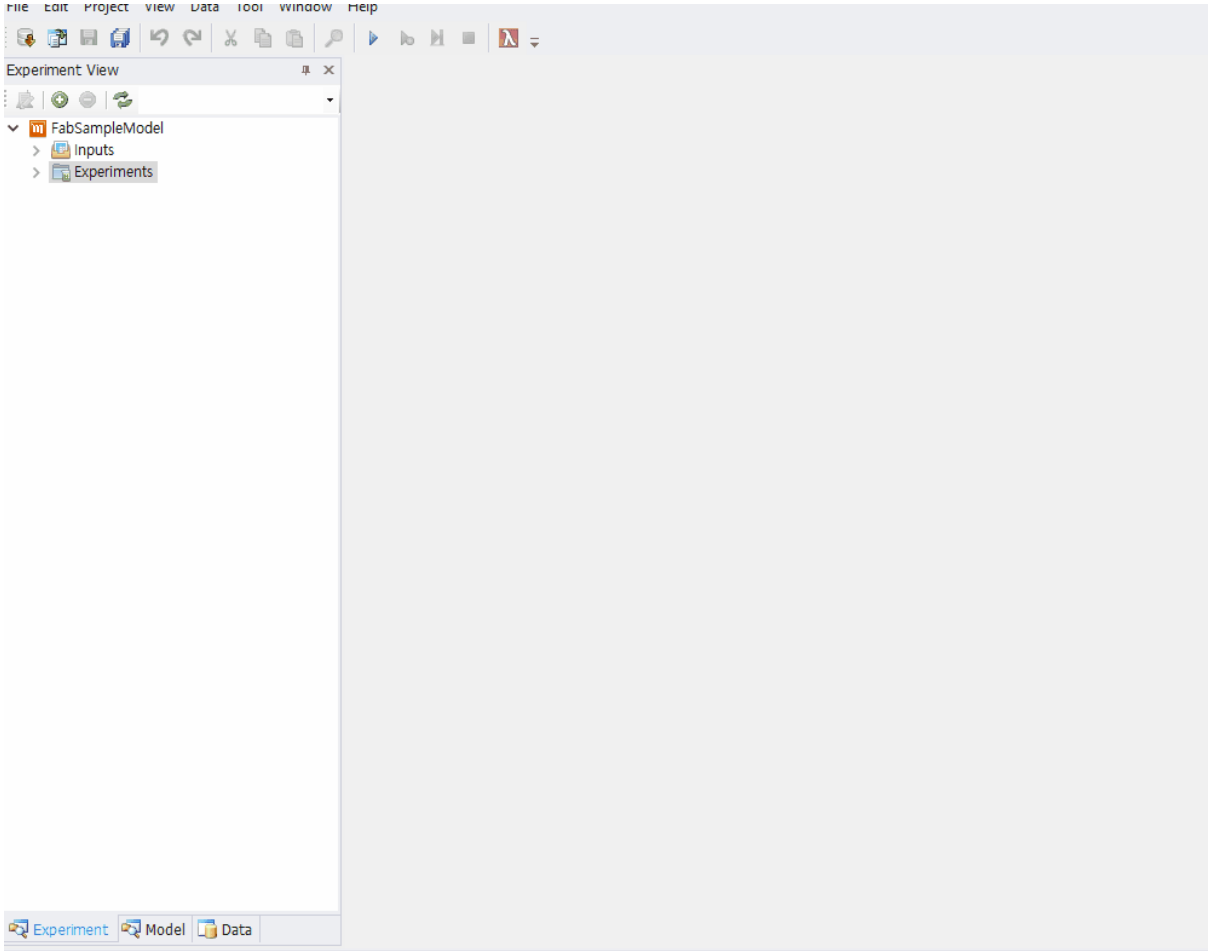
Connection String이 포함된 DataSource의 경우 DataSource를 별도로 Import/Export 해야 합니다.

DataItem Export 예시

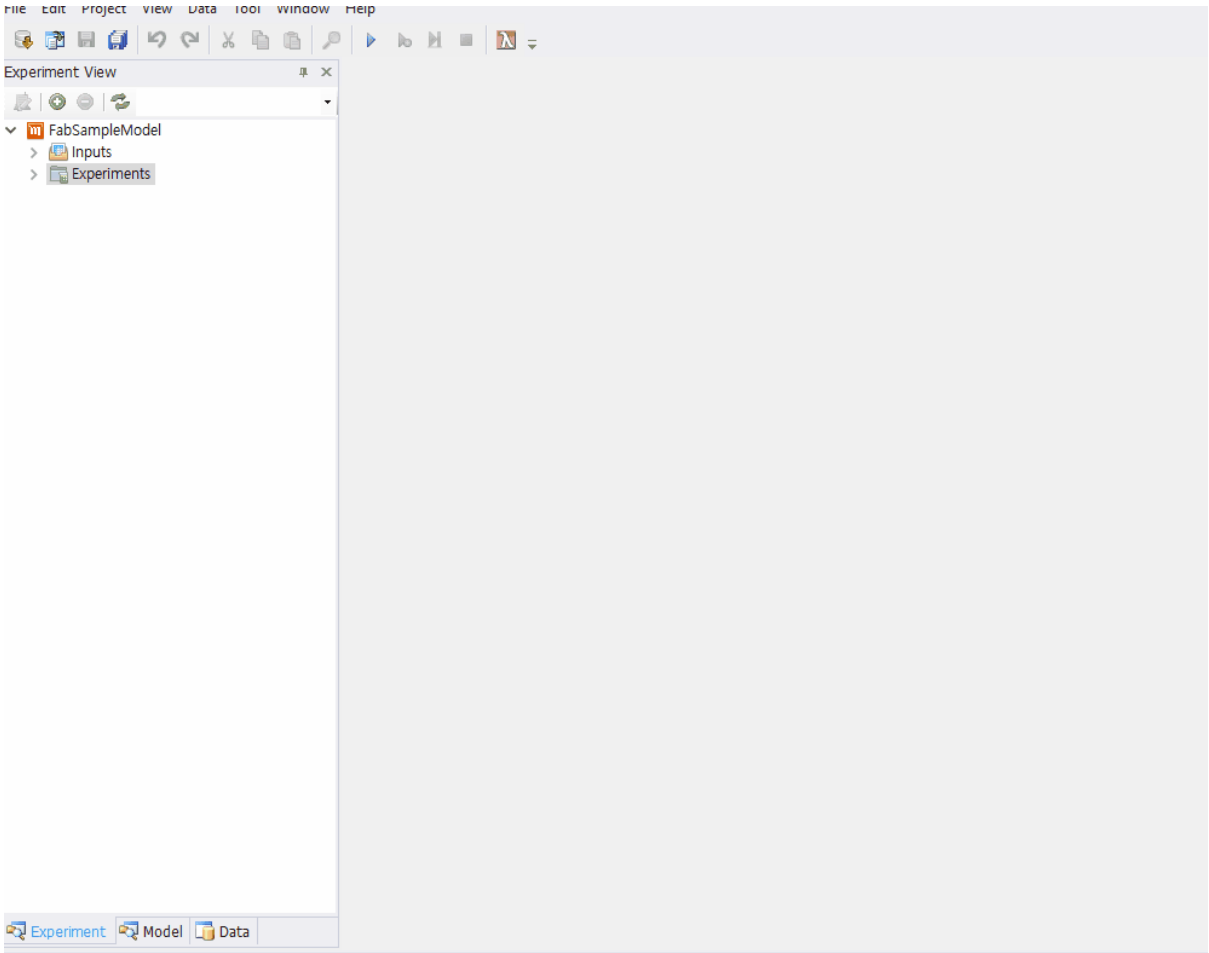
단일 DataItem Export하기



다중 DataItem Export하기

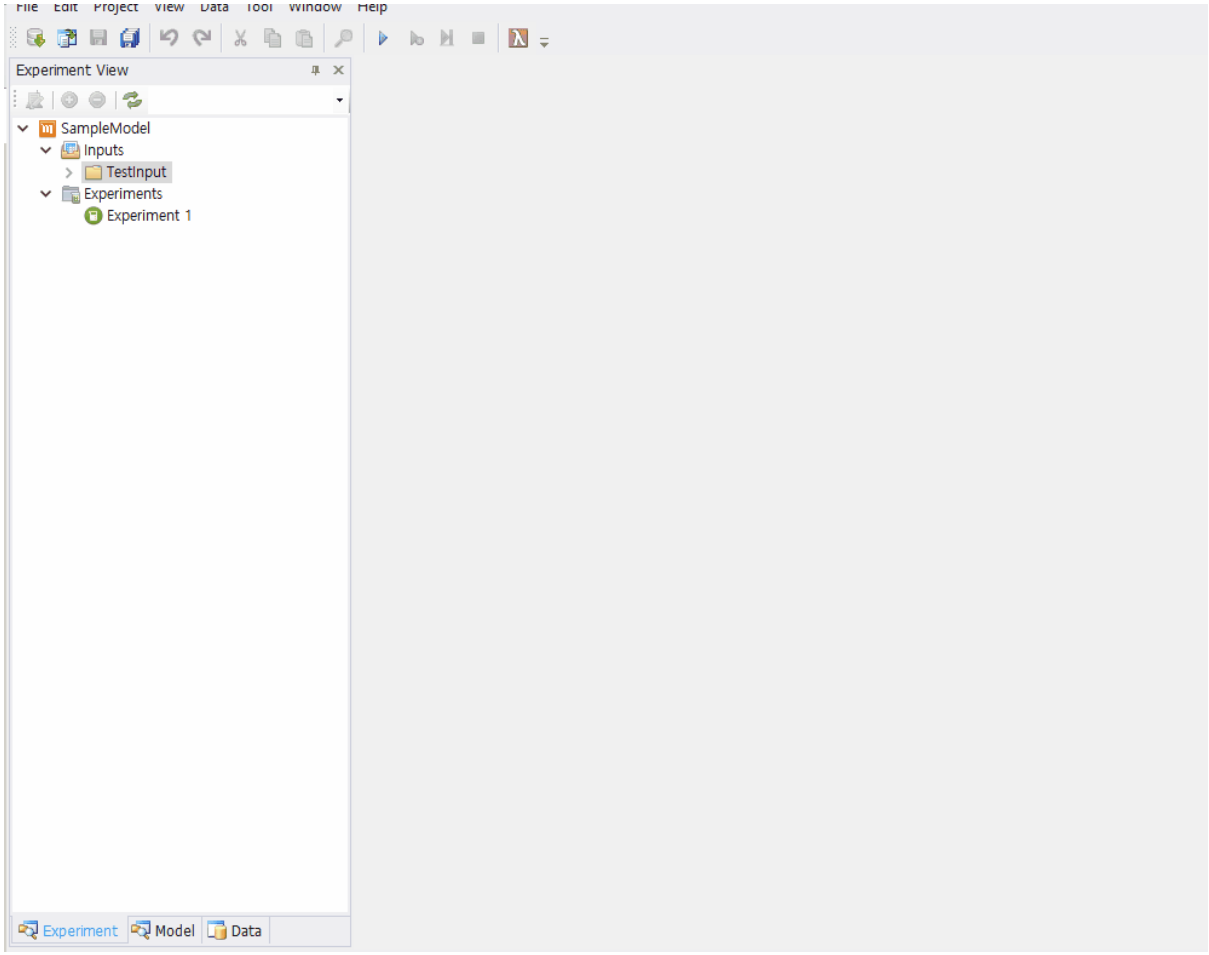


스키마 전체 Export하기

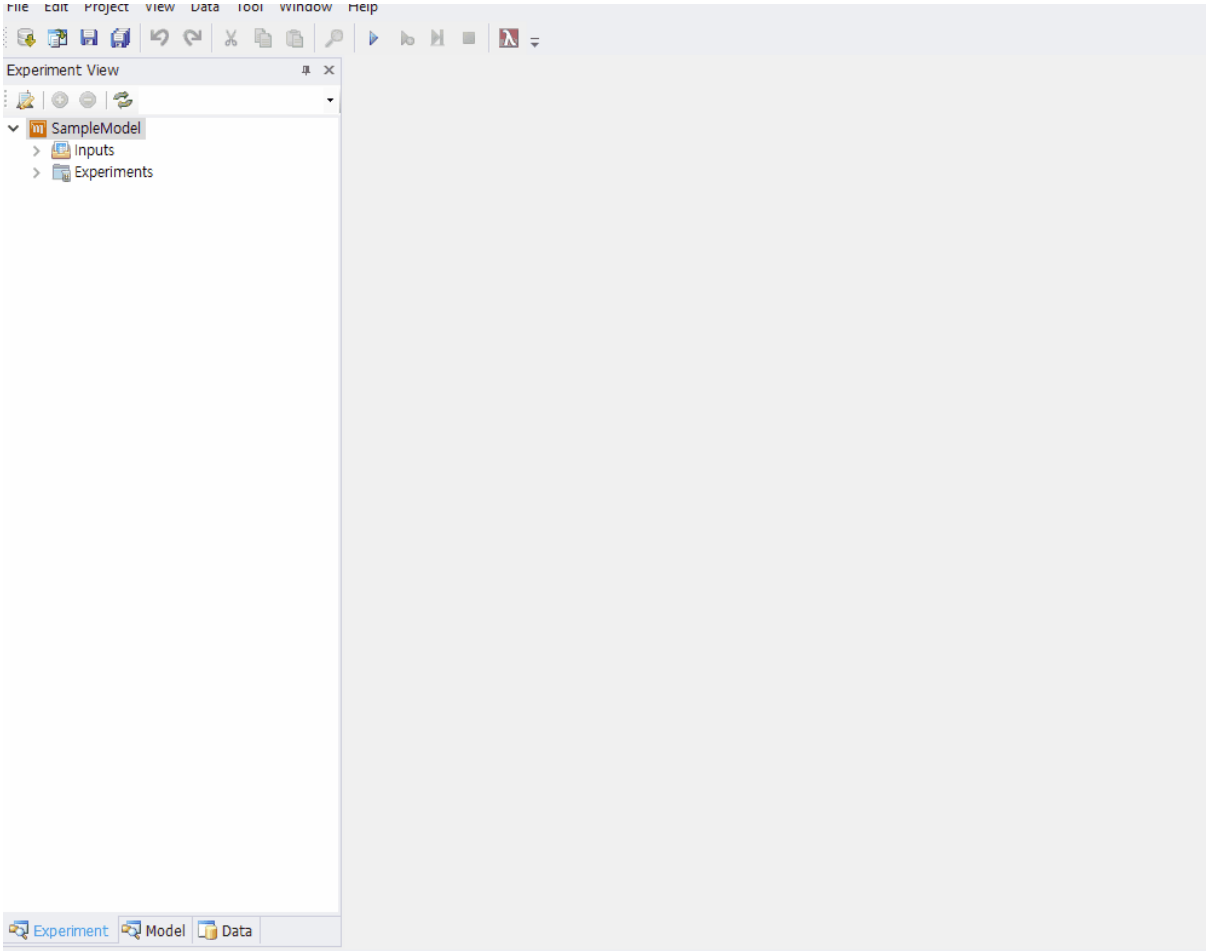


DataItem Import 예시

단일/다중 DataItem Import하기



스키마 전체 Import하기

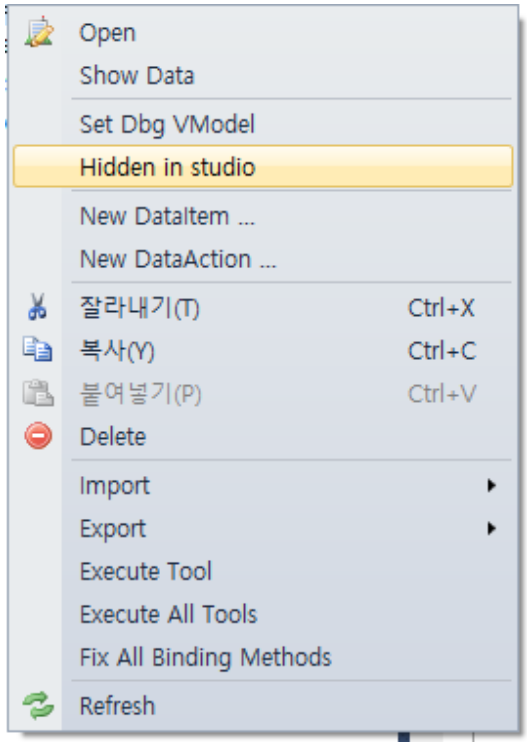


Mozart Studio에서 Export한 DataItem을 Mozart IDE에서 Import 하는 방법은 [DataItem 정보 Import/Export 하는 방법](#)을 참고 바랍니다.

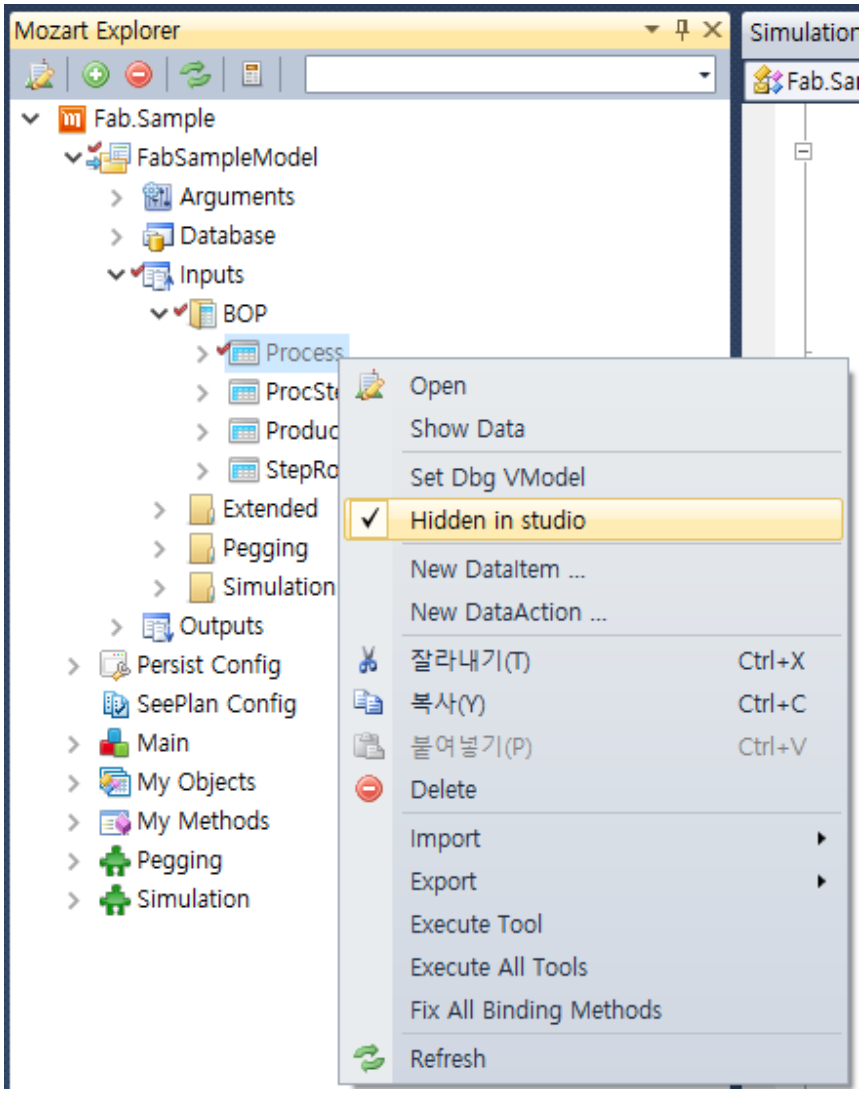
DataItem 숨기는 방법

MOZART IDE를 통해 특정 DataItemdm 숨기면서 MOZART Studio에 최종적으로 노출시킬 DataItem을 설정 할 수 있습니다. 기본적으로는 별도의 설정이 없으면 IDE를 통해 추가한 DataItem은 MOZART Studio에서 노출이 됩니다. 다음은 MOZART IDE상에서 DataItem을 숨기는 방법입니다.

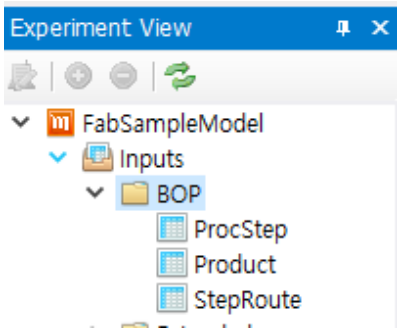
1. MOZART Explorer의 **[Model > Input/Output > DataGroup >DataItem]** 선택하고 마우스 우 클릭 메뉴에서 **[Hidden in studio]** 메뉴를 선택합니다.



2. 아래와 같이 **[Hidden in studio]**가 체크되어있고 해당 DataItem 명의 색상이 변경된 것을 확인하였으면 변경 사항을 저장합니다.

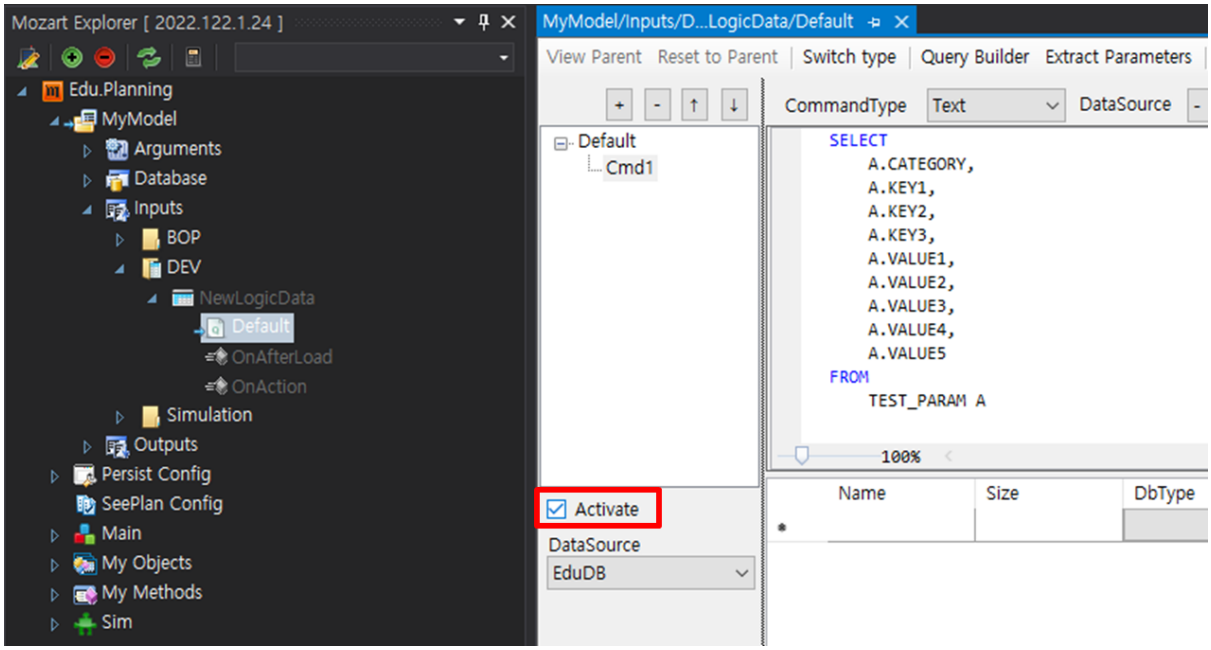


3. 프로젝트를 빌드하여 MOZART Studio를 실행합니다. DataItem을 숨긴 **[Model > Input/Output > DataGroup]**을 선택하여 숨겨진 DataItem이 안 보이는지 확인합니다.



주의 사항

MOZART Server 에서 모델의 DataItem이 **[Hidden In Studio]**를 통해 숨겨져도 DataAction이 연결되어 있으면 수행이 됩니다. DataAction 이 수행되지 않기를 원할 경우에는 **Activate**를 체크 해제하기 바랍니다.

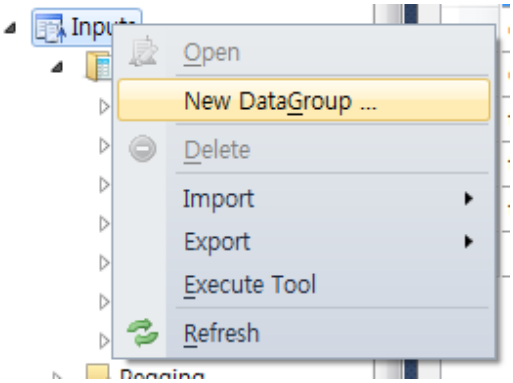


Schema 편집 방법

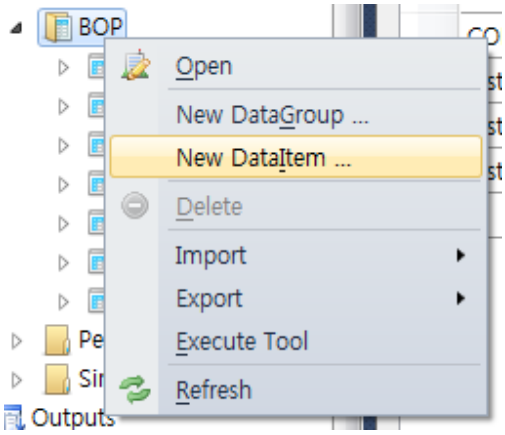
시스템에서 사용될 Input, Output Data 에 Schema 및 Schema 를 구성하는 Column 에 대한 속성을 설정하는 방법을 설명합니다. Input Data 의 경우 기본 Schema 편집 외에 추가적으로 InputMart 에 해당 Data 를 참조하기 위해 View 를 정의 할 수 있습니다. 아래는 두 가지 처리 방법을 설명합니다.

Schema 편집방법

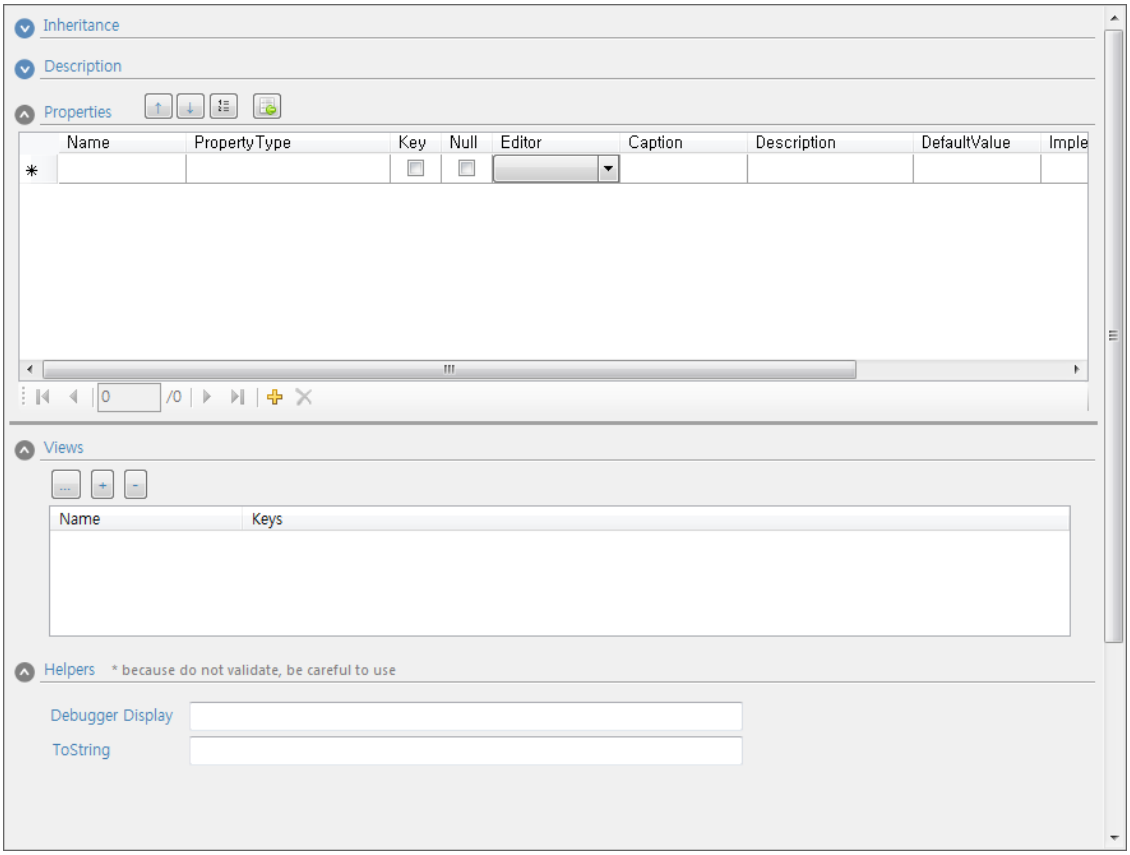
1. MOZART Explorer 의 **[Model > Inputs]** 를 선택하고 팝업메뉴에서 **[New DataGroup]**메뉴를 선택합니다.



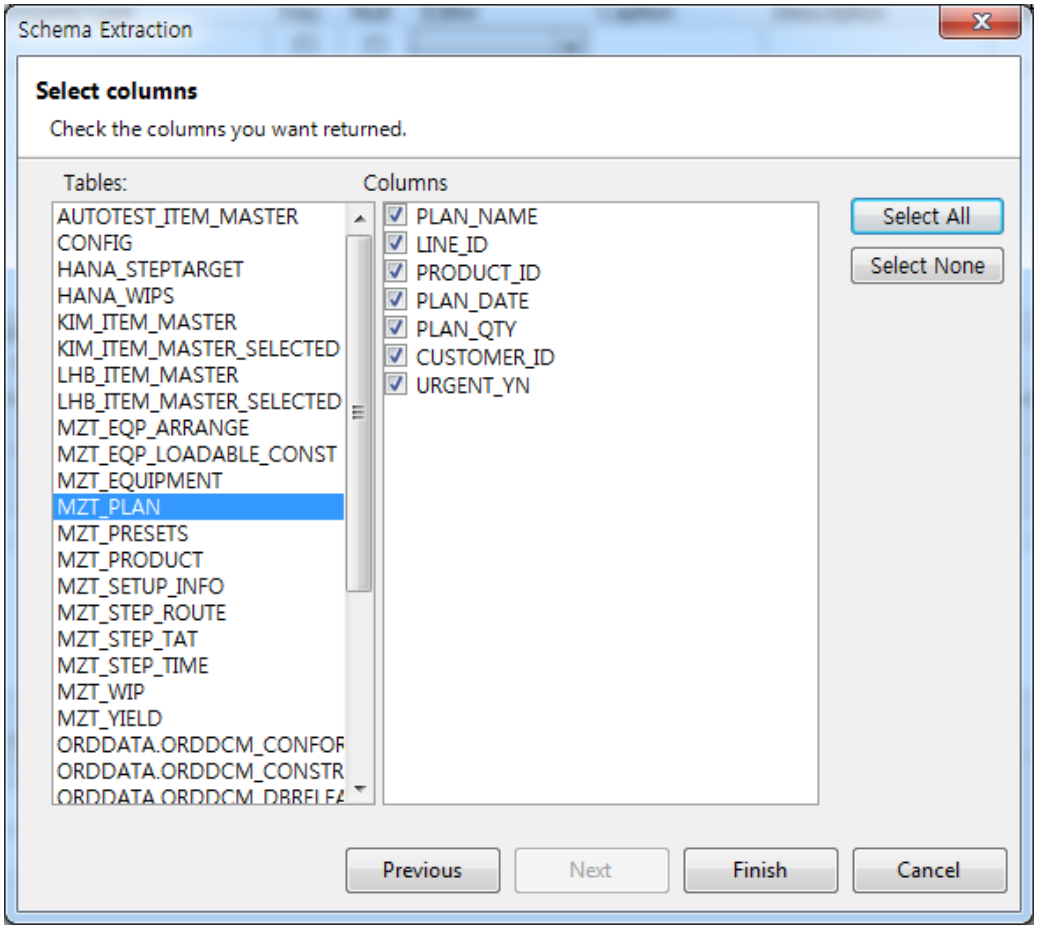
2. 새로 추가된 Data Group 이나 기존 Data Group 을 선택하고 팝업메뉴에서 [New DataItem]을 선택합니다.



3. DataItem 명을 입력하면 Tree 에 입력한 DataItem 이 생성되며 Schema 편집 화면이 열립니다.

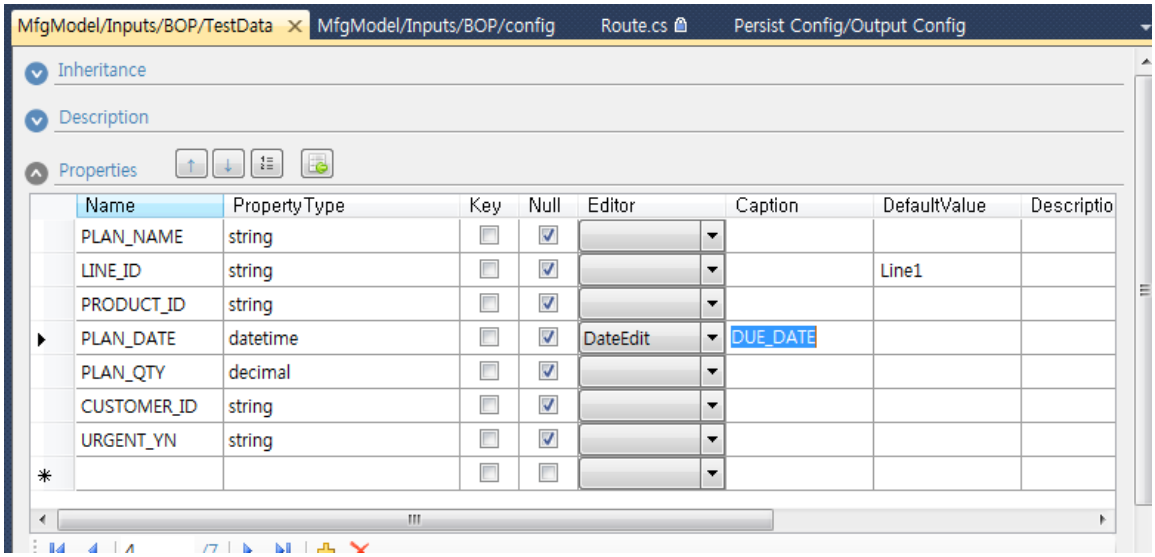


4. Data Column 을 입력하는데 Column 별로 수동 입력을 할 수도 있고, 특정 Data Source 의 Table 로부터 Column 을 가져와 Schema 를 생성할 수도 있습니다.
    - DB Table 로부터 생성하기 위해서는 상단의 [버튼이름] 버튼을 클릭합니다.
    - Schema Extraction Dialog 가 활성화되고 “Select a datababe” 콤보박스에서 기존에 연결정보 중 대상 Database 를 선택합니다.
    - 만일 대상 Database 가 없는 경우 신규로 연결정보를 입력합니다. (Datasource 입력 방식 참조)
    - Data 선택된 경우 **[Next]** 버튼으로 Table 및 Schema 에 사용될 Column 을 선택합니다.



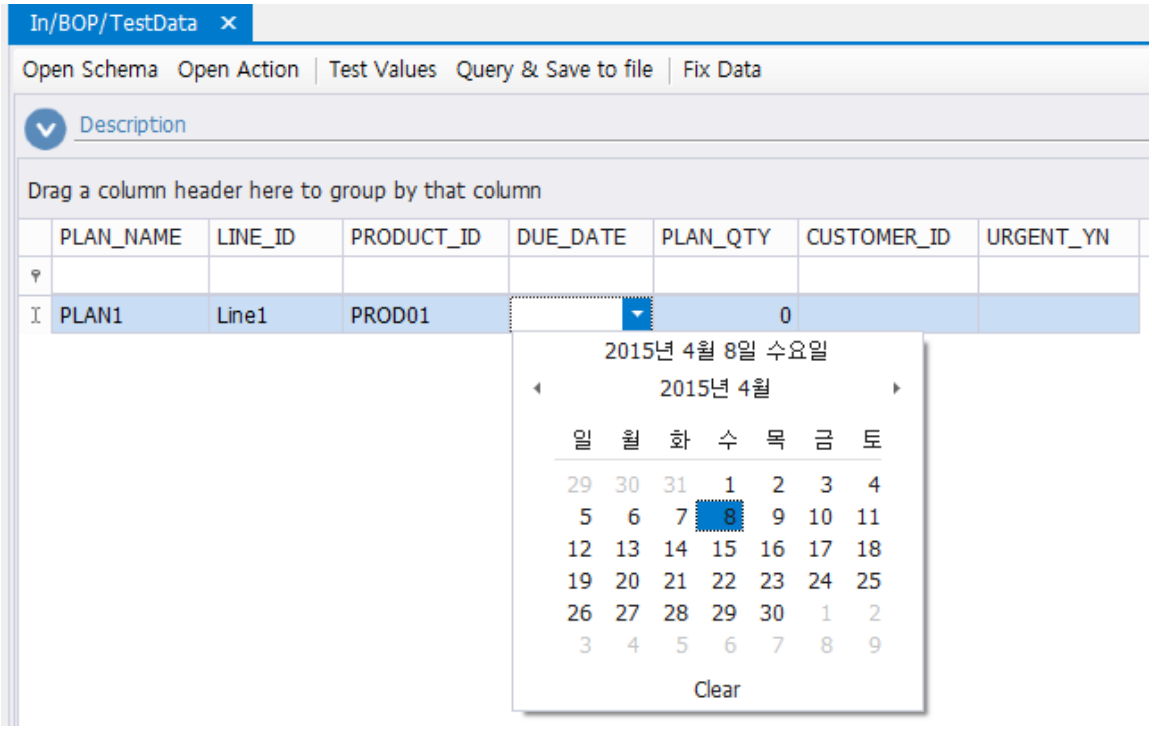
- Column 을 선택한 후 **[Finish]** 버튼을 클릭한 후 생성된 Schema 를 확인합니다.

5. Data Column 의 추가적인 속성을 입력합니다.

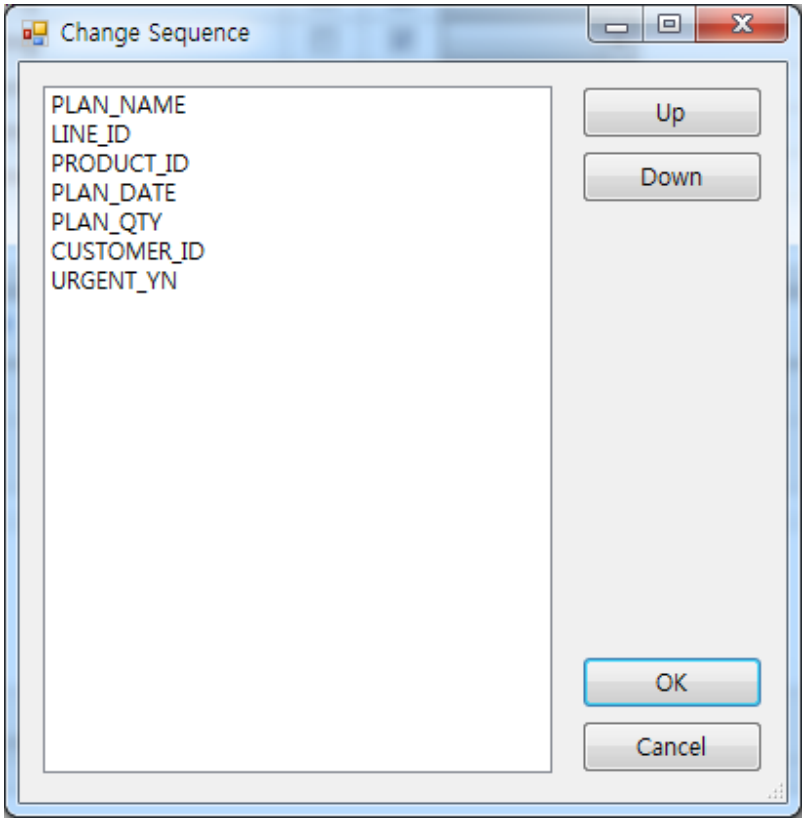


- **이름** : 열 명
- **PropertyType** : Column Data 타입

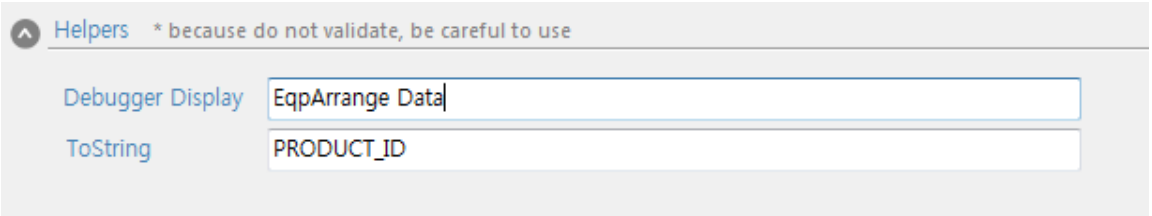
- **Caption** : MOZART Studio 상에서의 Column 표시 명입니다. 위 그림에서 PLAN_DATE 의 Caption 을 DUE_DATE 으로 설정한 경우 아래 그림과 같이 Studio 에서 그리드의 Column 명이 DUE_DATE 로 표시됩니다.
- **Description** : Column 설명
- **Key** : Column 의 Primary Key 여부입니다. 입력된 Data 에 대한 유효성을 체크합니다.
- **Null** : Column 의 Null 값 허용 여부입니다. 입력된 Data 에 대한 유효성을 체크합니다.
- **Editor** : MOZART Studio 의 Experiment 창에서 Data 그리드의 Column 값 입력시 입력 컨트롤로 유형은 {“TimeEdit”,”DateEdit”,”MemoEdit”,”ComboBox”}의 네 가지 유형이 있습니다. Editor 를 선택한 경우 Studio 에서 Data 를 입력할 때 선택한 Editor 가 활성화 됩니다. PLAN_DATE 에 대해 “DateEdit” 타입으로 선택한 경우 아래 그림과 같이 입력 시 날짜 선택 Dialog 가 활성화됩니다.
- **DefaultValue** : 기본값입니다. 상기 그림에서 LINE_ID 속성의 기본값을 “Line1” 으로 설정한 경우 아래와 같이 Data 입력 시 기본값으로 자동 입력됩니다.



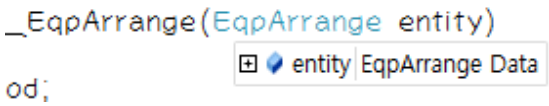
6. 입력한 Column 의 순서조정이 필요한 경우, 속성 그리드의 위치 조정 대상 속성을 선택한 후 ⬆️ ⬇️ 버튼으로 위치를 조정하거나, [버튼이름] 버튼으로 순서조정 다이얼로그박스를 사용하여 조정할 수 있습니다. 아래와 같은 다이얼로그를 통해 리스트박스에서 순서조정이 필요한 Column 을 선택한 후, 우측 상단의 **[Up][Down]** 버튼을 사용하여 위치를 조정하고 완료합니다.



7. **Helper 설정** : Debugging 및 구현 시점에 Data 클래스에 대한 표시명 및 기본 문자열을 설정합니다.

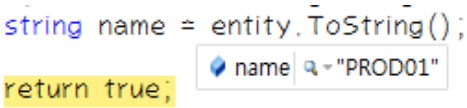


- **Debugger Display** : Debugging 시 변수에 대한 정보 표시 명입니다. 상기 이미지와 같이 설정하는 경우 Debugging 시 해당 객체는 아래 그림과 같이 표시됩니다.

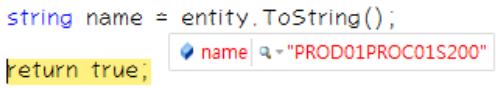


> **Note**
> Schema 에 정의된 속성값을 표시하기 위해서는 {LINE_ID}{PRODUCT_ID} 와 같은 방식으로 속성값을 대괄호"{}" 로 감싸서 입력합니다. 그외에 입력된 문자는 위의 예와 같이 입력값이 문자열로 그대로 출력됩니다.

- **ToString** : Debugging 시점 및 로직 작성 시점에 객체의 .ToString() 함수를 사용할 때의 결과로 사용할 속성을 설정합니다. Data 의 속성명을 설정하거나 간단한 Expression 을 사용할 수 있습니다. 상기 그림과 같이 설정한 경우 아래 왼쪽과 같이 결과가 출력되며, ToString 속성을 “PRODUCT_ID+PROCESS_ID+STEP_ID” 와 같이 설정하는 경우 아래 오른쪽 그림과 같이 Expression 결과가 출력됩니다.



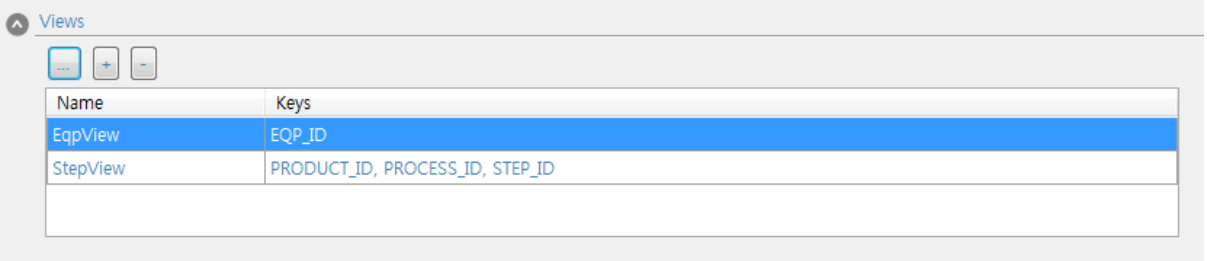
“PRODE_ID”로 설정한 경우



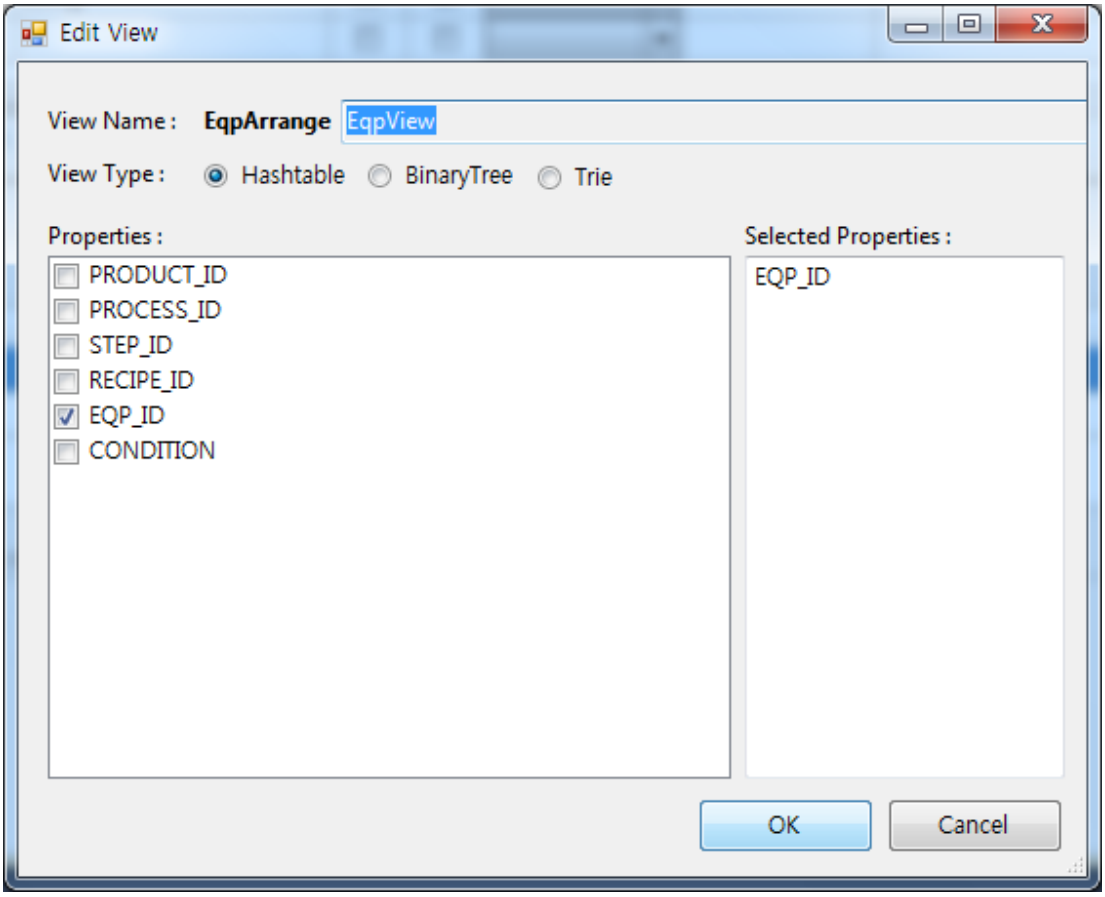
“PRODUCT_ID+PROCESS_ID+STEP_ID”로 설정한 경우

DataView 편집방법

Schema 로 정의된 Data 는 기본적으로 InputMart 에 EntityTable 형태로 자동으로 생성됩니다. Persist config 에서 “Use temporary context” 옵션을 설정한 경우 TempMart 에 생성됩니다. 시뮬레이션 수행 시 생성된 EntityTable 의 Data 를 사용할 때 검색속도를 향상시키기 위해 참조되는 Key 에 따라 View 를 설정할 수 있습니다. 다음은 Data 의 View 를 설정하는 방법을 설명합니다.



1. DataItem 편집 화면 하단의 View 섹션에서 […] 버튼을 클릭하면 View 설정창이 열립니다.



2. View 설정창에서 View 이름을 설정하고, View Type 을 선택한 후 해당 View 의 Key 항목을 선택한 후 [OK] 버튼을 눌러 저장합니다. Data 의 양이나 검색방식에 따라 View Type 을 선택합니다. Sorting 이 필요없이 검색만 수행하는 경우에는 HashTable type 을 주로 사용하며, Sorting 을 많이 해야 하는 경우에는 Binary Tree 를 사용하는 편이 보다 효율적입니다.
3. Lsit 에 입력한 View 가 생성됨을 확인합니다.
4. 해당 View 는 아래의 예제와 같이 InputMart 를 통해 참조할 수 있으며, EntityView 의 설정된 Key 로 빠른 검색이 가능합니다. 아래는 생성된 View 의 참조 및 활용에 대한 예제입니다.

View 사용 예제

```
public IList<string> GETLOADABLEEQPLIST(DispatchingAgent da, IHandlingBatch hb, ref bool
handled, IList<string> prevReturnValue)
{
   SiteLot lot = hb.Sample as SimpleMfgSemiconLot;
   var arrs = InputMart.Instance.EqpArrangeStepView.FindRows(lot.Product.ProductID, lot.
Process.ProcessID, lot.CurrentStepID);
   HashSet<string> loadables = new HashSet<string>();
   foreach (EqpArrange arr in arrs)
```

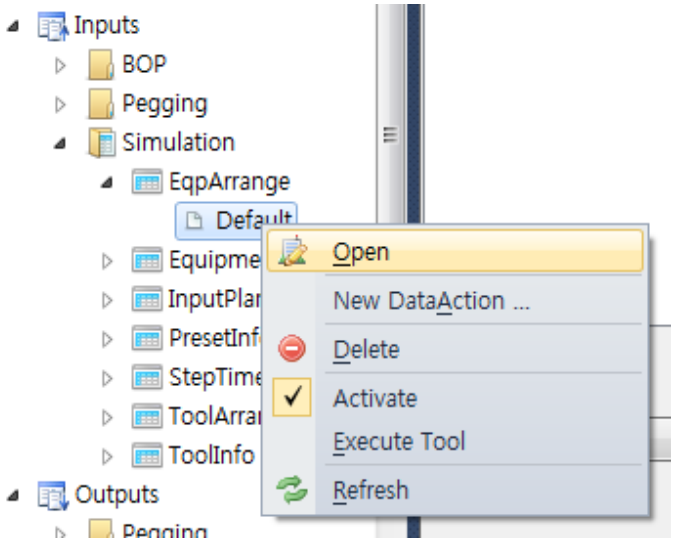
```
    {
        loadables.Add(arr.EQP_ID);
    }

  return loadables.ToList<string>();
}
```

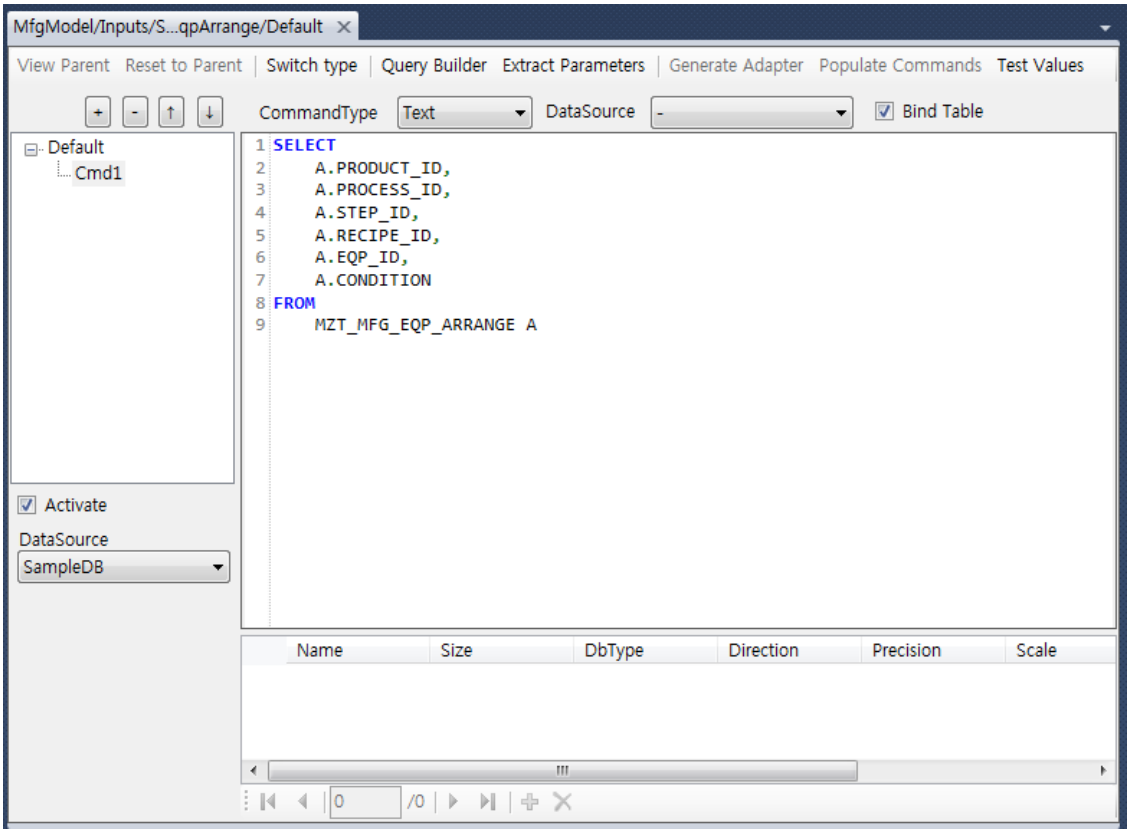
DataAction 편집 방법

DataAction 은 복수 개의 Query 로 구성될 수 있는 Query 관리단위입니다. Input DataItem 에 Binding 되는 DataAction 의 경우, Input Schema 에 Data 를 Download 하는 역할을 하는 Query 이며, Output DataItem에 Binding 되는 DataAction 의 경우 해당 Output Data 를 Database 에 삽입하기 위한 Query 입니다. DataAction 은 다음의 절차로 추가, 수정, 삭제할 수 있습니다.

DataAction 편집방법

1. 대상이 되는 DataItem 을 선택하고 Tree 를 펼치면 “Default” DataAction 이 표시되며, 해당 노드를 더블 클릭하거나 팝업의 **[Open]** 메뉴를 클릭하면 DataAction 편집창이 열립니다.





2. 만일에 기존의 DataAction 외에 신규 DataAction 을 추가하는 경우에는 해당 DataItem 을 선택하고 팝업메뉴의 **[New DataAction]** 을 사용합니다. 한 DataItem 에 DataAction 이 복수 개 등록된 경우에는 Activate 항목이 체크된 DataAction 이 실행됩니다.

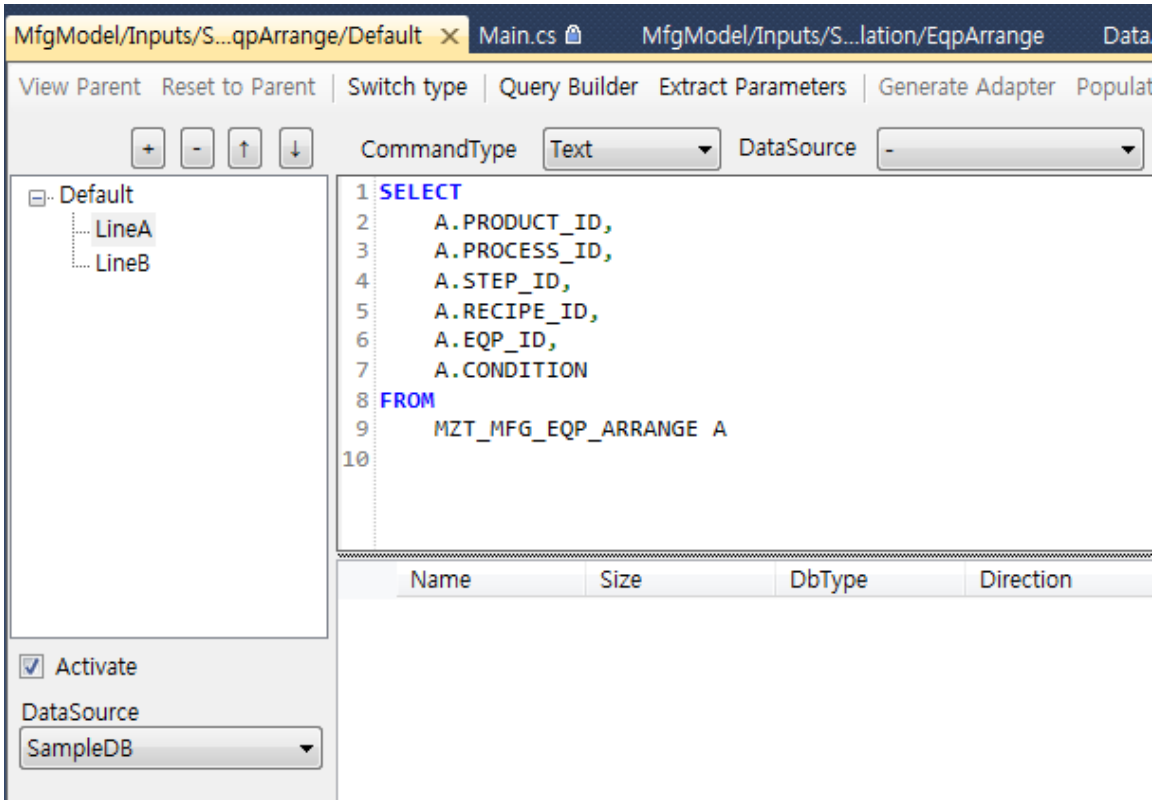
3. DataAction 의 왼쪽 하단에 Data Source 콤보박스에서 DataAction 의 Query 에서 사용할 Data Source 를 선택합니다.

4. 한 DataAction 은 여러 개의 Query 문장을 저장할 수 있습니다. 편집창 왼쪽 Tree 에 표시된 최하위 노드가 개별 Query 입니다. 상단의 ➕ ➖ 버튼을 통해 Command 를 추가할 수 있으며, ⬆️ ⬇️버튼을 통해 실행 순서를 조정할 수 있습니다. Tree 에서의 순서대로 실행됩니다.

복수 Command 등록 예제

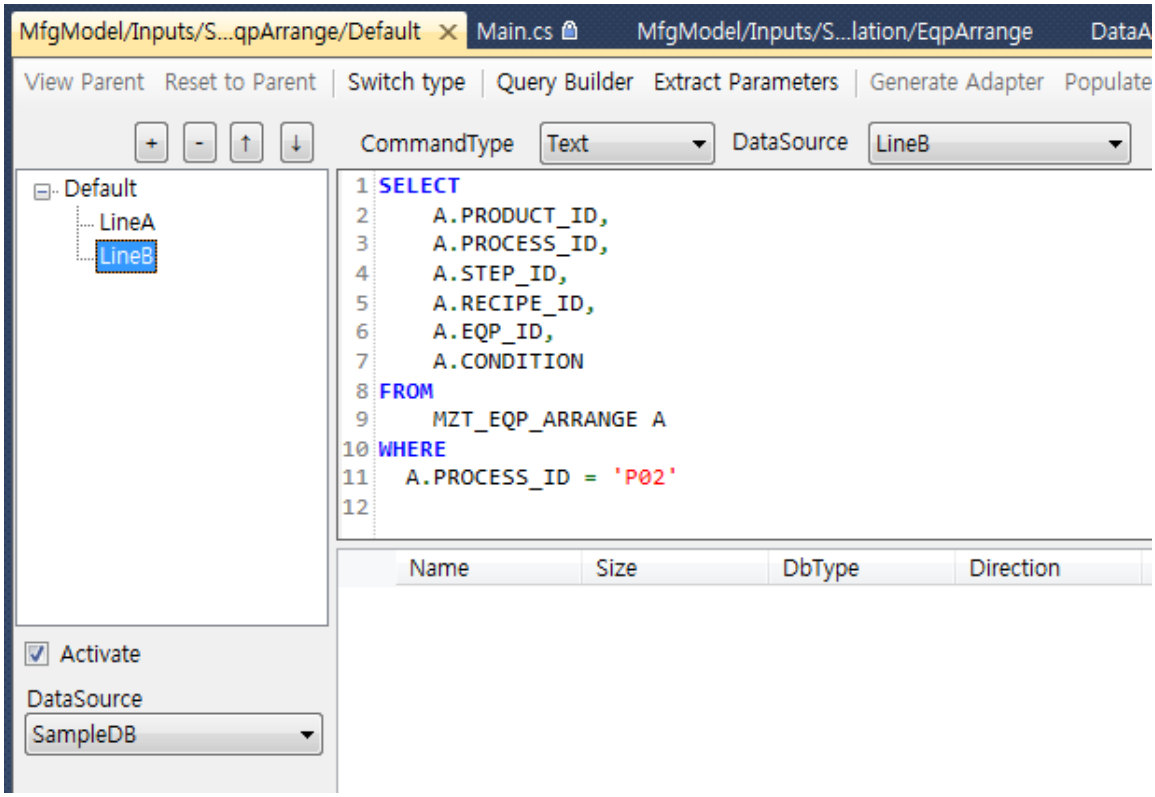
첫 번째 Command 는 SampleDB 에서 LineA 에 대한 Data 를 Query 하고, 두 번째 Command 는 LineB DB 에서 LineB 에 대한 Data 를 Query 하여 두 결과의 합집합으로 Input Data 를 처리하는 예제입니다.

[첫 번째 Command 등록]



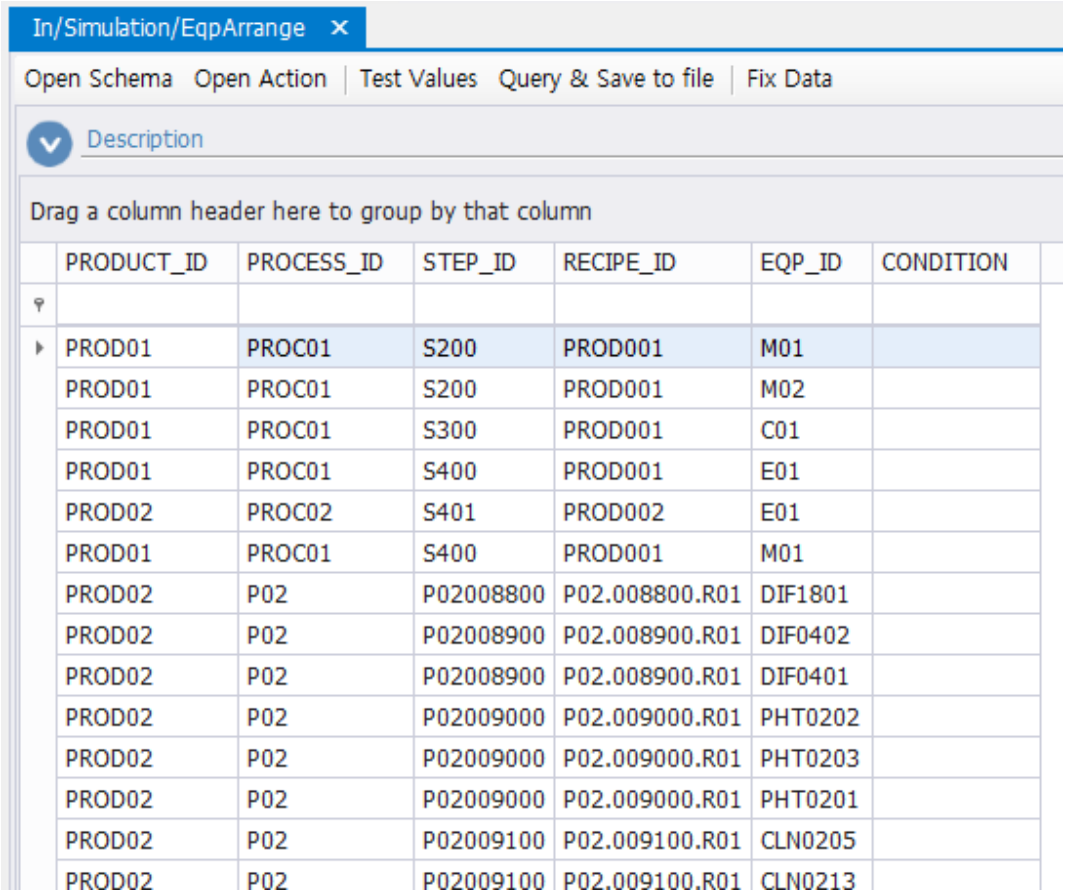
[두 번째 Command 등록]

Command 별 Data Source 를 두 번째 DB 에 대한 소스로 설정하고, Query 결과 Column 은 첫 번째 Command 와 동일하게 구성합니다.



[Query 결과]

[Query & Save to file] 메뉴로 실행하면 아래와 같이 두 Data 가 병합된 형태의 결과를 확인 할 수 있습니다



PRODUCT_ID	PROCESS_ID	STEP_ID	RECIPE_ID	EQP_ID	CONDITION
PROD01	PROC01	S200	PROD001	M01	
PROD01	PROC01	S200	PROD001	M02	
PROD01	PROC01	S300	PROD001	C01	
PROD01	PROC01	S400	PROD001	E01	
PROD02	PROC02	S401	PROD002	E01	
PROD01	PROC01	S400	PROD001	M01	
PROD02	P02	P02008800	P02.008800.R01	DIF1801	
PROD02	P02	P02008900	P02.008900.R01	DIF0402	
PROD02	P02	P02008900	P02.008900.R01	DIF0401	
PROD02	P02	P02009000	P02.009000.R01	PHT0202	
PROD02	P02	P02009000	P02.009000.R01	PHT0203	
PROD02	P02	P02009000	P02.009000.R01	PHT0201	
PROD02	P02	P02009100	P02.009100.R01	CLN0205	
PROD02	P02	P02009100	P02.009100.R01	CLN0213	

5. 오른쪽 편집창에 Query 를 입력합니다.

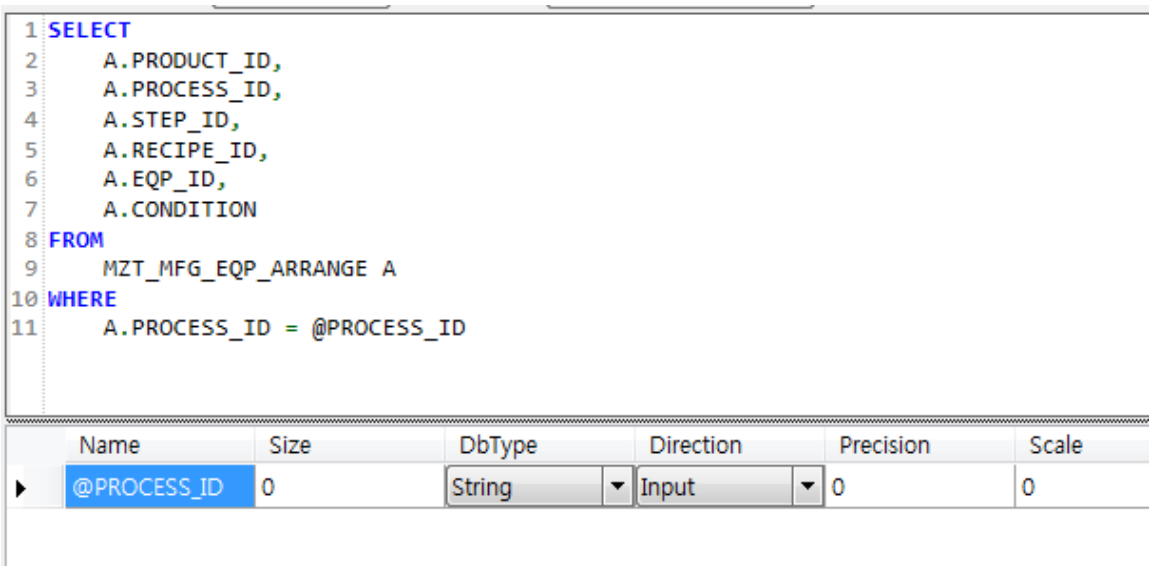
> **Note**
> 편집창 상단의 콤보박스 및 체크박스 옵션은 다음과 같은 설정을 지원합니다.

- **CommandType** : Query 편집창의 Query 종류 선택
  - **Text** : 일반 Query 문장을 입력합니다.
  - Stored Procedure : 저장프로시저 실행을 위한 프로시저 이름을 입력합니다.
  - **TableDirect** : Data Source 의 특정 Table Data 와 직접 연결
  - **DataSource** : Data Action 의 기본 Data Source는 왼쪽 TreeView 하단의 Data Source 가 되며, 만일 Tree 의 특정 Command 에서 사용할 Data Source 를 별도로 지정하고 싶은 경우 설정합니다. 해당 Command(Query) 에서는 지정

된 Data Source 에서 Query 를 실행합니다. 예를 들어 두 개의 다른 Data Source(A, B DB) 에서 동일한 Schema 의 Data 를 Query 하여 Data 의 합집합을 사용하는 경우 사용할 수 있는 방법입니다.

- **Bind Table** : 일반적으로 Output DataItem 에서 사용하는 옵션입니다. 해당 옵션이 선택된 경우 대상 Data Row 단위로 해당 Query 가 실행됩니다. Select Query 에는 반영되지 않습니다.

6. Query 작성 시, Arguments 를 사용하고자 하는 경우에는 “@” 문자를 접두어로 붙여 변수로 사용할 수 있습니다. Query 작성 시, 변수를 사용하여 작성한 후 상단 메뉴 바의 **[Extract Parameters]** 버튼을 클릭하면 하단의 그림과 같이 Parameters 가 자동 등록됩니다



위 그림에서 처럼 “@PROCESS_ID” 변수를 사용하여 입력된 PROCESS_ID 에 대한 ARRANGE 만 가져올 수 있도록 Query 를 작성할 수 있습니다. 설정된 Parameters 는 Main Control 의 **[Main > SetQueryArgs]** 함수에서 값을 설정할 수 있습니다. 또한 Studio 에서 사용자가 Query 를 테스트할 때는 Studio 의 **[Test Values]** 메뉴를 사용하여 Parameters 값을 설정할 수 있습니다. 아래는 두 가지 방식에 대한 사용 예제입니다.

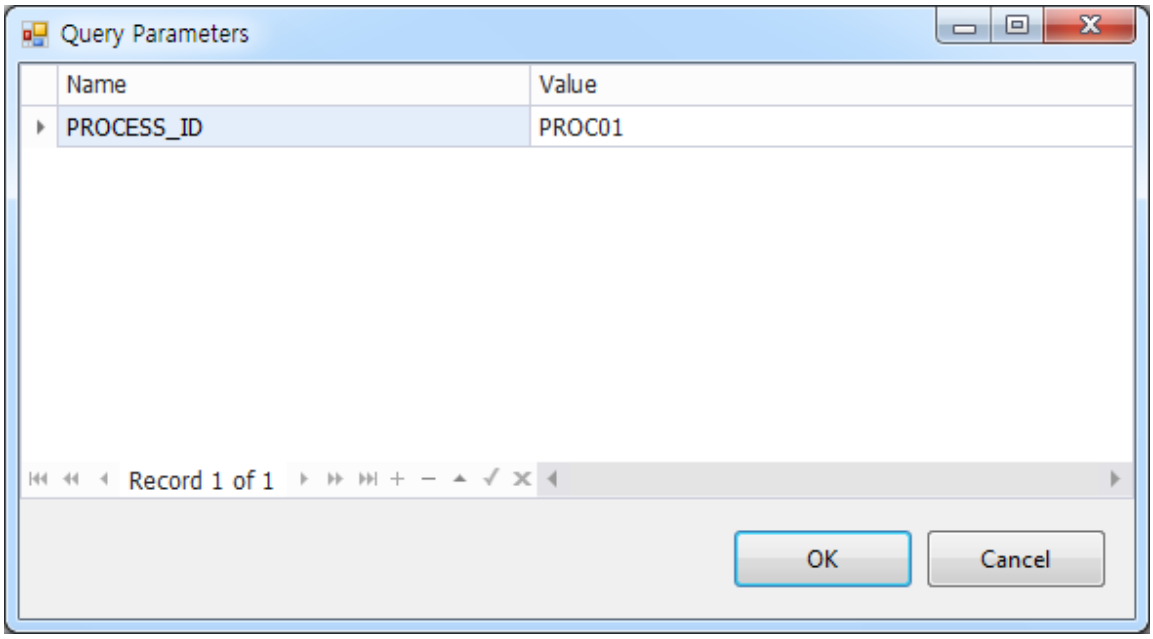
Query Arguments 의 설정 및 실행 결과

```
// Main>Main>SetQueryArgs 구현 내용입니다.
// 아래와 같이 변수를 입력하면, 입력 Data 의 Query 수행 시 Argument 를 반영하여 Query 합니다.
 public void SETUPQUERYARGS_0(ModelTask task, ModelContext context, ref bool handle
d)
 {
     // ModelContext 의 QueryArgs 에 Query 작성 시 사용한 변수에 대한 값을 입력합니다.
     context.QueryArgs.Add("PROCESS_ID", "PROC01");
 }
```

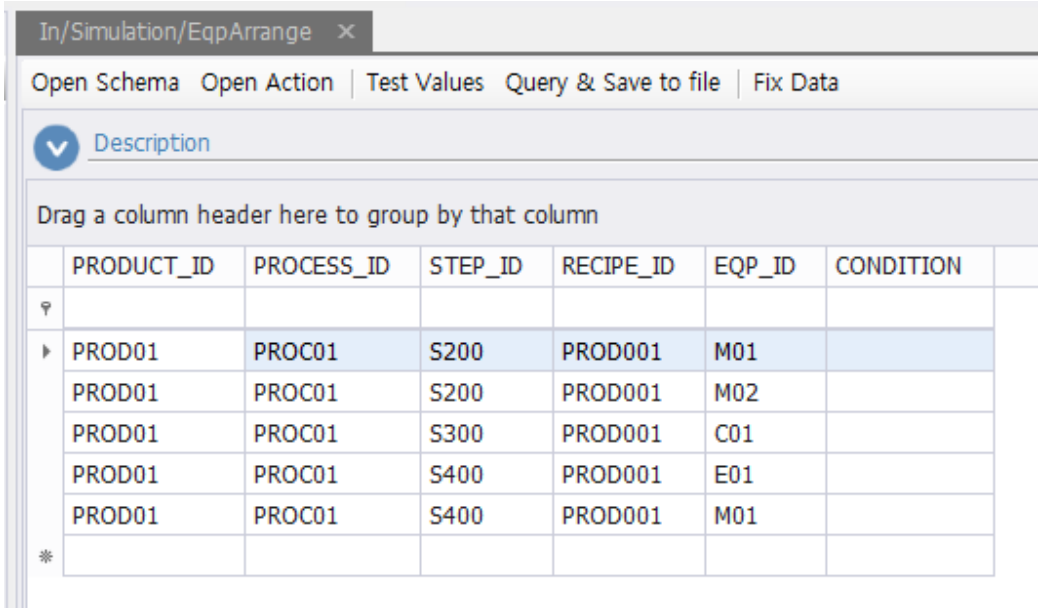
Studio 에서 테스트하기

Studio 상에서 해당 Query 를 테스트하기 위해서는 먼저 Input Data 를 조회한 후 상단의 **[Test Values]** 메뉴를 사용해서 Query 에 사용한 변수에 대한 값을 입력합니다. 그 후 **[Query & Save to file]** 메뉴를 사용하여 Query 를 수행하면 변수가 반영된 결과를 얻을 수 있습니다.

[Test Values 실행 및 변수값 입력]



[Query & Save to file 실행 결과]



7. 왼쪽 Query Tree View 하단의 Activate 체크 박스를 체크하고 저장하면 해당 Schema 에 대해 설정한 DataAction 이 활성화됩니다.
8. 상단 메뉴 바의 경우, dataaction 의 Command 관리 기능과 Query 를 지원하는 기능으로 구성됩니다.
   - **Query Builder** : Query 작성을 지원합니다. Data Source 에 연결한 후 Table Schema 를 이용한 Select, Insert, Update, Delete Query 생성을 지원합니다.
   - **Extract Parameter** : Query 에 삽입된 Parameters 를 추출하여 Query 편집창 하단의 Parameters List 에 자동으로 추가합니다. 추출대상이 되는 변수는 접두어 "@" 를 붙여 정의합니다.
     - 예: @LINE_ID, @PROCESS_ID 등
   - **Test Values** : Query 작성 시 Parameters 를 설정한 경우에 Parameters 의 테스트 값을 입력하기 위해 사용합니다. 입력된 값은 Studio 에서 **"Query & Save to file"** 메뉴 실행 시 기본적으로 사용됩니다. Studio 의 **"Test Values"** 메뉴를 실행하면 DataAction 에서 설정한 값이 기본으로 설정된 것을 확인할 수 있습니다.

참고. Stored Procedure 호출하는 방법

[CommandType] 중 `StoredProcedure` 타입의 경우 기능 오류가 있어 `Text` 타입으로 사용하는 것을 권장합니다.

데이터베이스별 사용 방법

MS SQL

MS SQL 에서는 실행 명령(`execute`, `exec`)을 통하여 실행할 수 있습니다. 실행 명령 뒤에 **데이터베이스이름.스키마이름.저장프로시저이름** 형태로 쿼리 작성이 되어야 합니다.

```
//매개변수 없을 때
exec Database_Name.Schema_Name.Procedure_Name

//매개변수 있을 때
exec Database_Name.Schema_Name.Procedure_Name @Param1, @Param2, ...
```

Oracle / MySql / DB2 / PostgreSQL

Oracle, MySql, DB2, PostgreSQL의 경우는 `Call` 명령을 통하여 실행할 수 있습니다. 매개변수와 관련하여 괄호를 저장 프로시저 이름 옆에 추가하여 쿼리 작성이 되어야 합니다.

```
//매개변수 없을 때
call Procedure_Name()

//매개변수 있을 때
call Procedure_Name(@Param1, @Param2, ...)
```

⚠️ Oracle의 경우 Call 명령은 Oracle 9i 버전부터 사용 가능합니다.
9i 이전 버전의 경우 `Exec(Execute)` 명령을 사용하기 바라며, `Exec(Execute)` 명령 사용시 매개변수가 없을 때는 저장 프로시저 이름 옆에 괄호를 사용하지 않습니다.

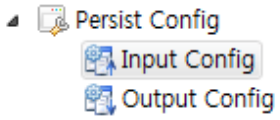
주의사항

Output DataAction에서 저장 프로시저를 호출하는 경우

- **Bind Table**의 옵션이 True인 경우 Data Row 갯수 만큼 쿼리가 수행되며, 의도에 맞게 수행되는지 설정 확인을 하시기 바랍니다.
- Output Data Row 값을 매개변수로 전달하여 프로시저로 저장하는 경우에는 저장 속도가 빠른 **BulkCopy 사용**을 권장합니다.

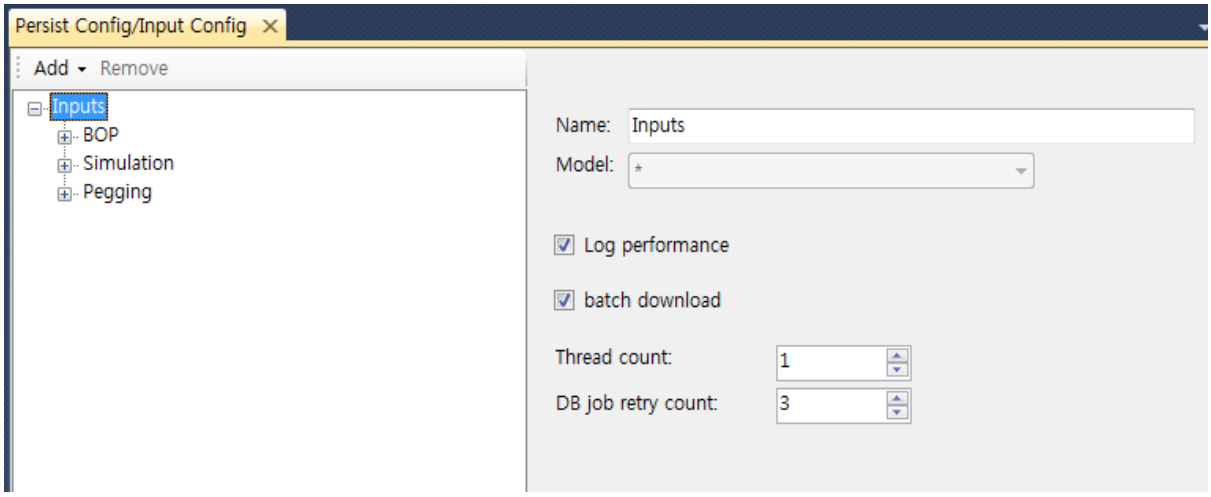
Input Persist Configuration 정의

Persist Config 에서는 Input DataItem 을 Memory 상의 Data 로 로딩하는 작업 절차와 방식을 정의합니다. Input Persist 의 구성요소는 Input Data Group, Input DataItem, Input Save Log 세 가지이며, 전체 Input Loading 시의 옵션은 Persist Config 의 Root Item 에 대한 입력을 통해 설정할 수 있습니다. 실행 메뉴는 MOZART Explorer 의 [Persist Config > Input Config] 입니다.



Input Data 전체 로딩 옵션 설정

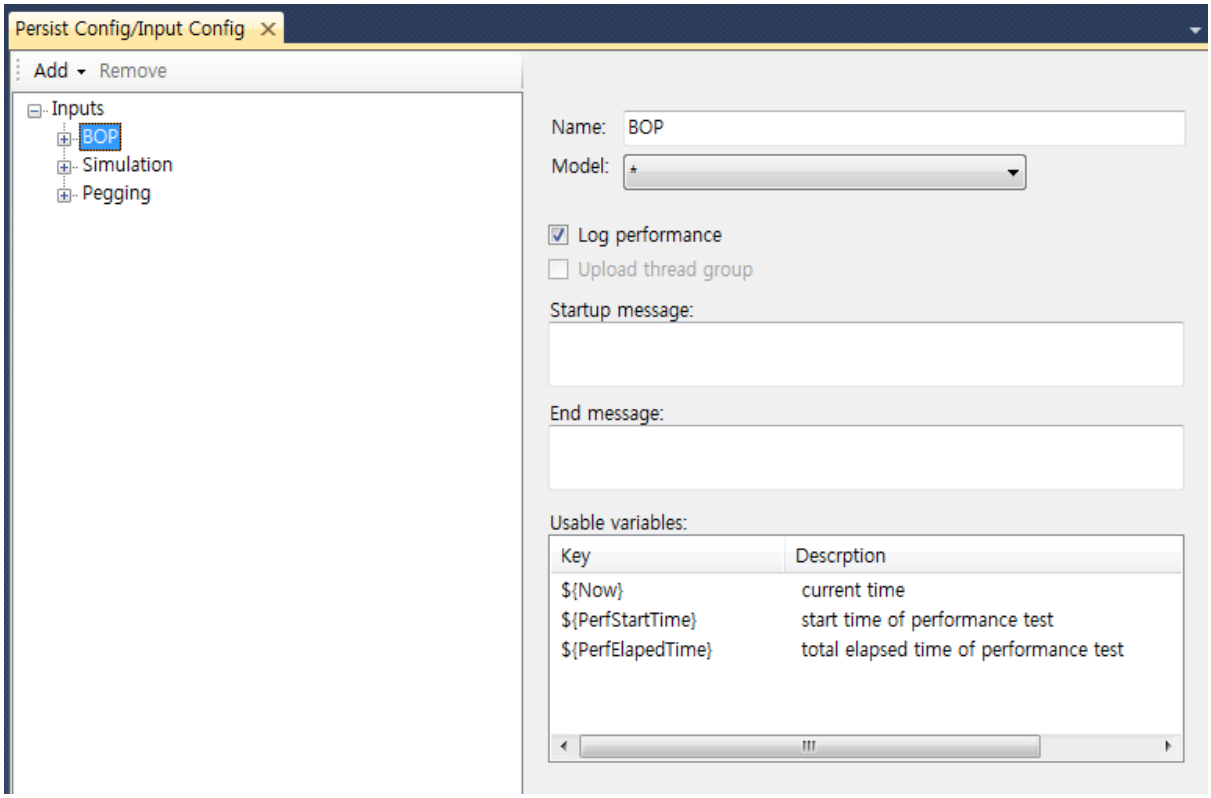
최상위 Tree 노드를 선택하면 Data 로딩을 위한 옵션을 정의할 수 있는 메뉴가 활성화됩니다.



- **Name** : Inputs 전체에 대한 Log 명
- **Model** : 해당 Group 의 설정 내용을 적용할 Model 을 선택합니다. “*” 로 설정하는 경우 모든 Model 에 적용되며, 특정 Model 을 지정하는 경우 해당 그룹 내의 모든 DataItem 에 대한 로딩은 지정된 Model 에서만 Data 를 올리도록 처리합니다.
- **Log performance** : 전체 Data 로딩 소요시간을 기록할지 여부를 설정합니다.
- **Tread count : Batch Download** 를 수행하는 경우 사용할 Tread 수를 지정합니다.
- **DB job retry count : Batch Download** 시 DB 연결, 명령의 재실행 횟수를 지정합니다.

Input Data Group 로딩 옵션 설정

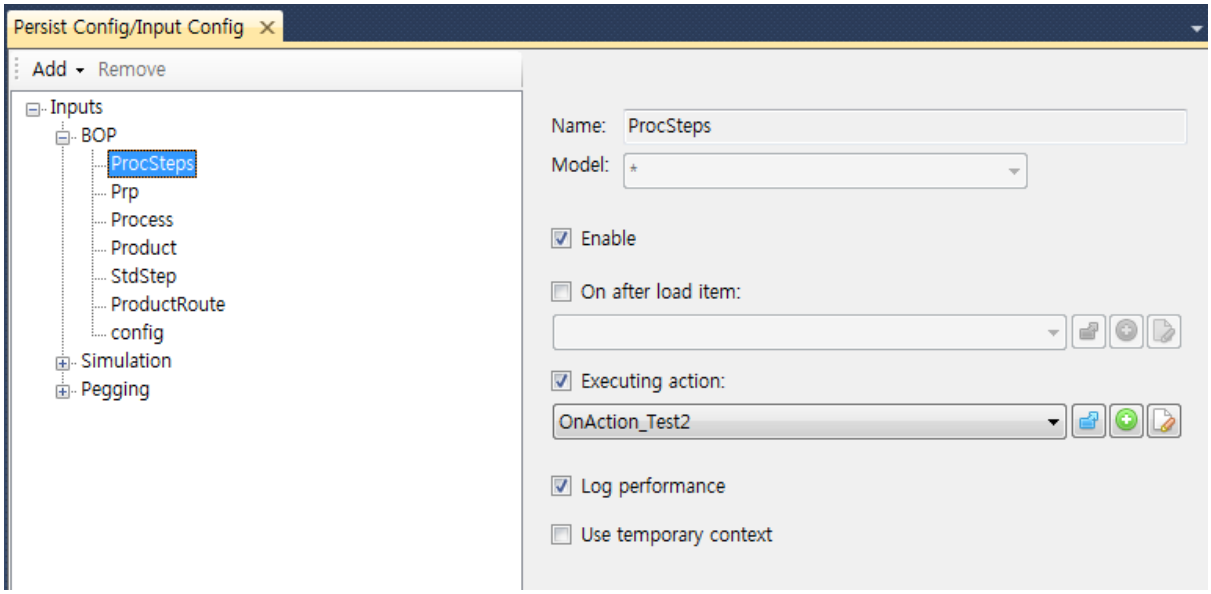
Data Group 별 로딩 옵션을 정의할 수 있습니다. Tree 메뉴에서 Data Group 을 선택하면 우측의 속성창을 통해 속성설정이 가능합니다.



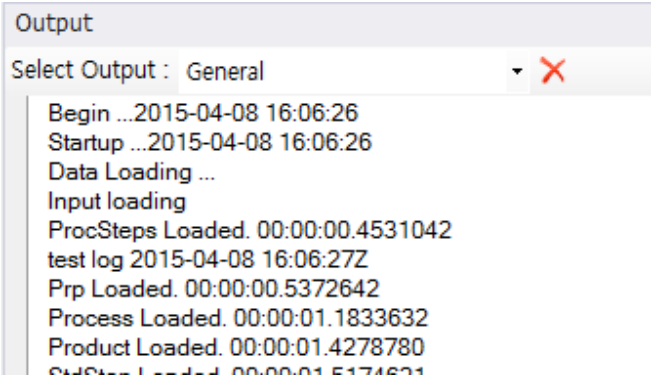
- **Name** : 로딩 시 사용할 Group 명입니다.
- **Log performance**: 전체 Data 로딩 소요시간을 기록할지 여부를 설정합니다.
- **Startup message**: Group 에 대한 로딩 시작 시점에 Log 에 남길 메시지를 편집합니다. 메시지 편집 시 하단의 Usable variables 를 사용할 수 있습니다.
    - **AtBegin** : 동일 시간 발생 Event 들을 처리하기 전에 가장 먼저 처리하는 옵션
    - **AtEnd** : 동일 시간 발생 Event 들을 모두 처리한 후 처리하는 옵션
- **End message**: Group 에 대한 로딩 완료 시점에 남길 메시지를 편집합니다. 메시지 편집 시 하단의 Usable variables 를 사용할 수 있습니다.

Input DataItem 별 로딩 옵션 설정

개별 DataItem 별로 로딩 시 옵션을 정의할 수 있습니다.



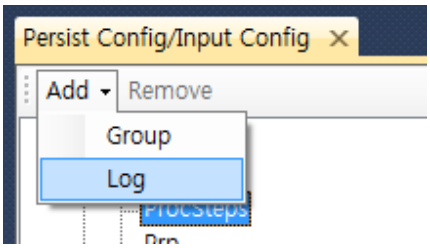
- **Name** : 로딩 시 사용할 DataItem 명으로 변경이 불가합니다.
- **Enable**: Data 로딩 여부를 설정합니다. 옵션을 체크하지 않은 경우 DataItem 이 자동으로 로드되지 않습니다.
- **On after load item** : DataItem 의 각 데이터 Row 별로 데이터의 변환이나 Validation 등을 사용자가 처리할 수 있는 Method 를 실행합니다. 본 항목이 체크된 경우 실행할 Method 는 바로 하단의 버튼을 이용하여 다음과 같은 작업을 수행할 수 있습니다. 함수의 리턴값은 해당 Data Row 의 로드 여부를 의미합니다. true 인 경우 해당 Row 가 로드된 Data 에 포함되며, false 인 경우 Filtering 됩니다. 메시지 편집 시 하단의 Usable variables 를 사용할 수 있습니다.
    - **Action 선택** : 사전에 등록된 Load Method 를 콤보박스에서 선택할 수 있습니다.
    - **Action 추가** : 해당 Data 를 가공하기 위한 함수명과 빈 함수의 소스를 생성합니다. [버튼이름] 버튼을 사용합니다. 함수가 생성되는 파일 명은 PersistInputs.cs 파일이며. Logic 폴더 하위에 생성됩니다.
    - **Action 편집** : 콤보박스에 선택된 함수를 편집하기 위해 소스로 이동합니다. [버튼이름] 버튼을 사용합니다.
    - **Action name 편집** : Action 의 함수 명을 편집하기 위해 사용합니다. [버튼이름] 버튼을 사용합니다.
- **Executing action**: DataItem 의 모든 Data 가 다 로딩 처리 된 후에 별도의 Load 처리를 추가적으로 하려고 하는 경우 Method 를 지정하고 이를 실행시키기 위한 옵션항목입니다. On afer load item 에서 Action 생성관련 메뉴를 동일하게 사용할 수 있습니다.
- **Log performance**: DataItem 의 로딩 시간 기록여부를 설정합니다. Check 하면 아래그림과 같이 Log 정보에 DataItem 에 대한 로딩 시간이 기록됩니다.

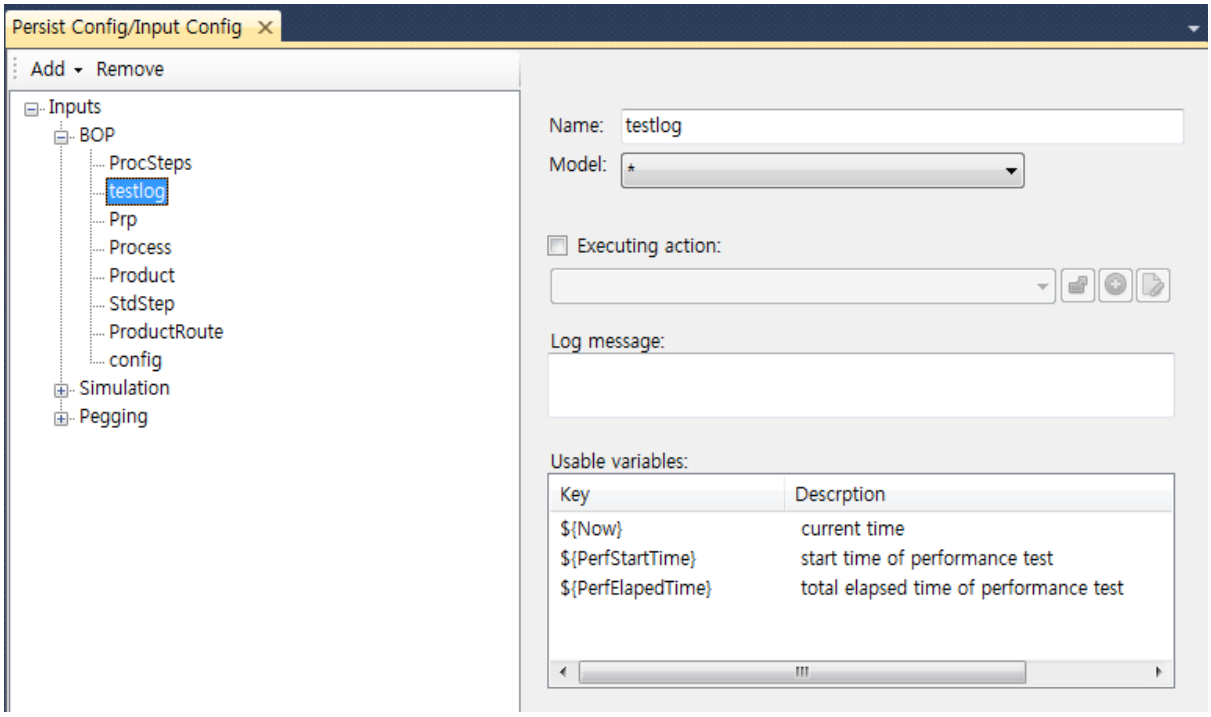


- **Use temporary context**: 일반적으로 DataItem 은 Model 의 실행 주기 내내 Memory 에 로드되어야 하는 Data 와 Data 로딩 시점에 임시적으로 로드했다가 Data 로드가 끝나는 시점에 Memory 에서 삭제해도 상관없는 경우가 있습니다. 본 옵션으로 이를 컨트롤 합니다. 본 옵션이 선택된 경우에는 Input Data Loading 이 완료되는 시점에 Data 가 Memory 에서 삭제됩니다. 생성된 클래스는 TempMart 를 통해 참조가 가능합니다.

일반 Log 생성 옵션 설정

Log Item 은 Data 로딩 시, DataItem 과 관계없이 특정 위치에 Log 를 남기거나 특정 Method 를 실행시키려고 할 때 사용할 수 있습니다. Log Item 은 Persist Config 의 [Add > Log] 메뉴를 통해 추가할 수 있습니다. Tree 특정 노드를 선택한 후 Log 를 추가하는 경우에는 해당 노드의 바로 아래 Log Item 이 생성되며, 속성 페이지를 통해 Log Item 의 옵션을 설정할 수 있습니다.

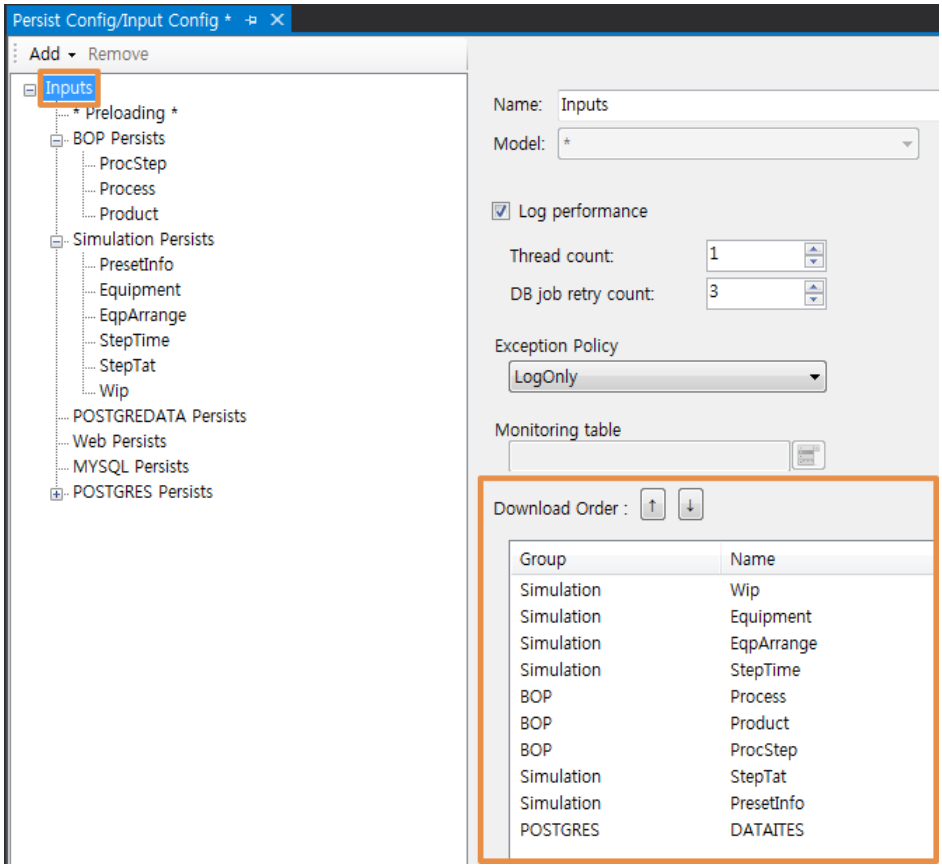




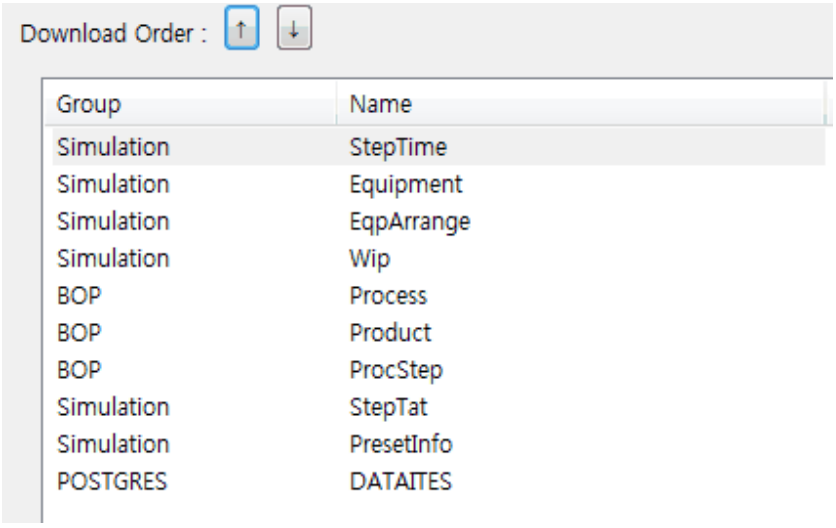
- **Name** : Log 명을 입력합니다.
- **Model** : 해당 항목을 사용할 Model 을 지정합니다.
- **Executing action** : Log 실행 순서에 실행할 Method 를 지정하고 실행 여부를 설정합니다. 본 옵션이 설정된 경우에는 하단에 등록된 Method 가 실행됩니다. 사전에 등록된 Method 를 찾아서 설정하거나, 신규 Method 를 생성 할 수 있습니다.
- **Log message** : Log Item 의 위치에 남길 Log 메시지를 설정합니다. Log 메시지 본문은 Usable variables 에 등록된 Key 를 사용하여 편집이 가능합니다.

Input Data Download 순서 조절 기능

개별 DataItem간 Database로부터 데이터 다운로드하는 순서를 조정할 수 있습니다.



다운로드 순서는 [Download Order]에서 DataItem을 선택하여 또는 버튼을 누르면서 순서를 변경 할 수 있습니다.



```
Start database downloading...
Download 'StepTime' table ... WorkQueue[0] : 00:00:10.7492213
Download 'Equipment' table ... WorkQueue[0] : 00:00:10.0326172
Download 'EqpArrange' table ... WorkQueue[0] : 00:00:10.0081284
Download 'Wip' table ... WorkQueue[0] : 00:00:10.0384283
Download 'StepTat' table ... WorkQueue[0] : 00:00:10.0169663
Download 'PresetInfo' table ... WorkQueue[0] : 00:00:10.0274868
Download 'ProcStep' table ... WorkQueue[0] : 00:00:10.0076164
Download 'Product' table ... WorkQueue[0] : 00:00:10.0080015
Download 'Process' table ... WorkQueue[0] : 00:00:10.0070665
 Total elapsed time = 00:01:31.1491454
```

Before

```
Start database downloading...
Download 'ProcStep' table ... WorkQueue[0] : 00:00:10.7319680
Download 'Product' table ... WorkQueue[0] : 00:00:10.0167083
Download 'Process' table ... WorkQueue[0] : 00:00:10.0073193
Download 'Wip' table ... WorkQueue[0] : 00:00:10.1127932
Download 'StepTime' table ... WorkQueue[0] : 00:00:10.0274185
Download 'Equipment' table ... WorkQueue[0] : 00:00:10.0287525
Download 'EqpArrange' table ... WorkQueue[0] : 00:00:10.0075813
Download 'StepTat' table ... WorkQueue[0] : 00:00:10.0182964
Download 'PresetInfo' table ... WorkQueue[0] : 00:00:10.0277260
 Total elapsed time = 00:01:31.2536983
```

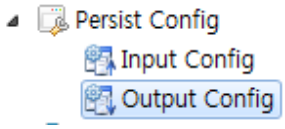
After

DB Download 순서 변경 전후 모습

데이터 다운로드 순서와 Persist 단계에서 메모리에 데이터 적재하는 순서는 다르며, 데이터 적재 순서는 좌측 Tree 메뉴에서 순서를 조정합니다.

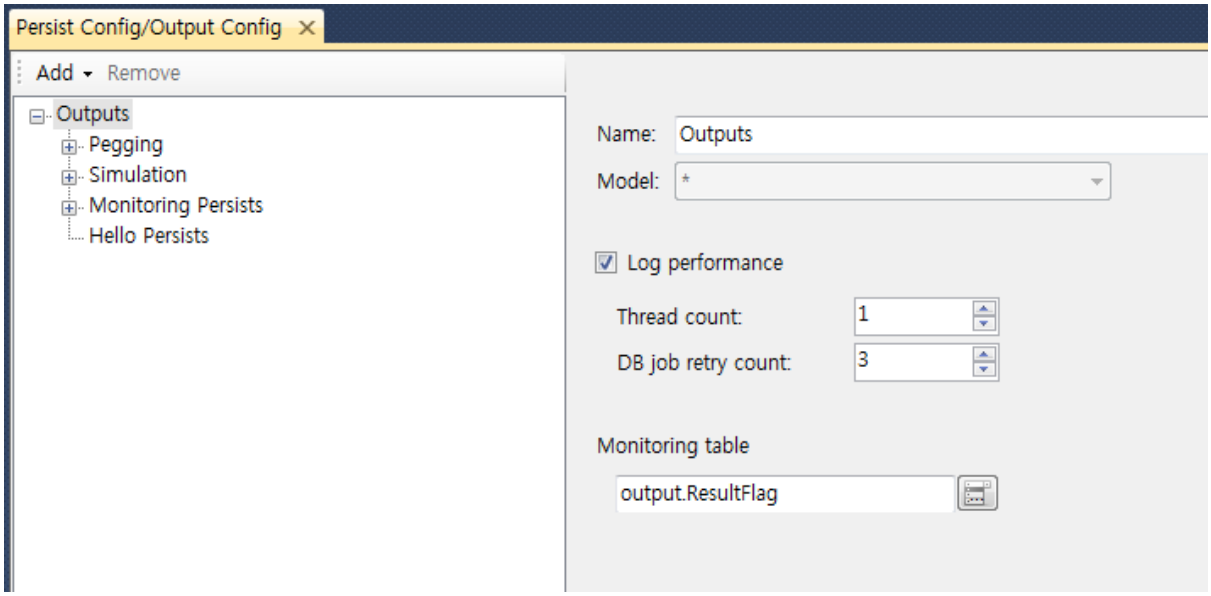
Output Persist Configuration 정의

output persist configuration 에서는 Memory 상의 Output DataItem 을 File, DB 에 저장하는 작업절차와 방식을 정의합니다. 일련의 과정을 진행하기 위해 Output Persist 를 정의하는 구성요소는 Output Data Group, Output DataItem, Output Save Log 의 세 가지 유형으로 나눌 수 있으며, 전체 Output 을 처리하기 위한 옵션도 Persist Config 를 통해 설정할 수 있습니다. 실행 메뉴는 [Persist Config > Output Config] 입니다.



Output Data 전체 저장 옵션 설정

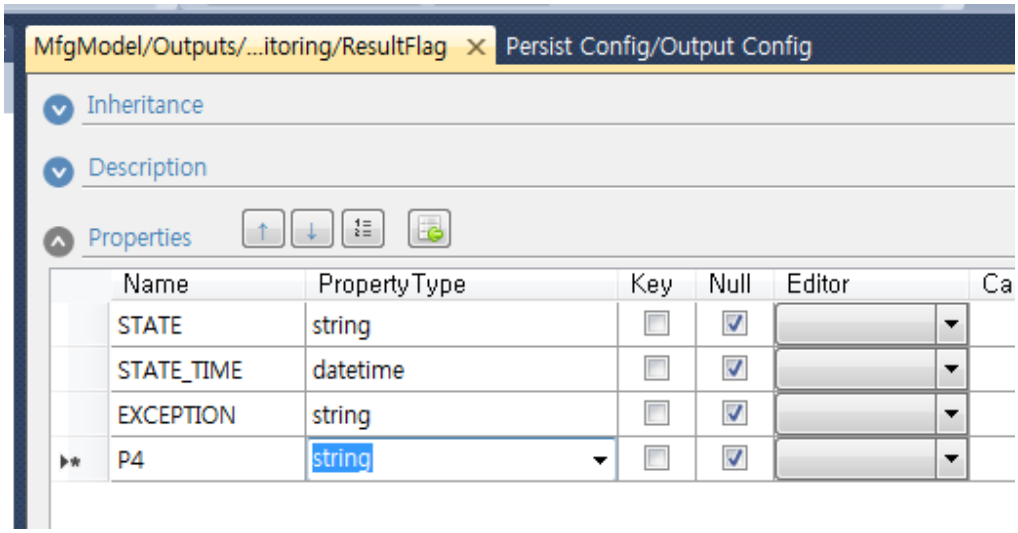
최상위 Tree 노드를 선택하면 Data 저장을 위한 옵션을 정의할 수 있는 메뉴가 활성화됩니다.



- **Name** : Outputs 전체에 대한 Log
- **Log performance**: 전체 Data 저장 소요시간을 기록할지 여부를 설정합니다.
- **Thread count: Batch Upload** 를 수행하는 경우 사용할 Thread 수를 지정합니다.
- **DB job retry count**: DB 연결, 명령의 재실행 횟수를 지정합니다.
- **Monitoring table** : 전체 Task 의 수행 종료 시점에 Task 수행 성공/실패 여부를 기록하기 위한 Output 을 정의할 수 있습니다. Output 을 설정한 후 Main Control 의 ShutDown Action 을 사용하여 결과를 기록할 수 있습니다.

Monitoring 사용 예시

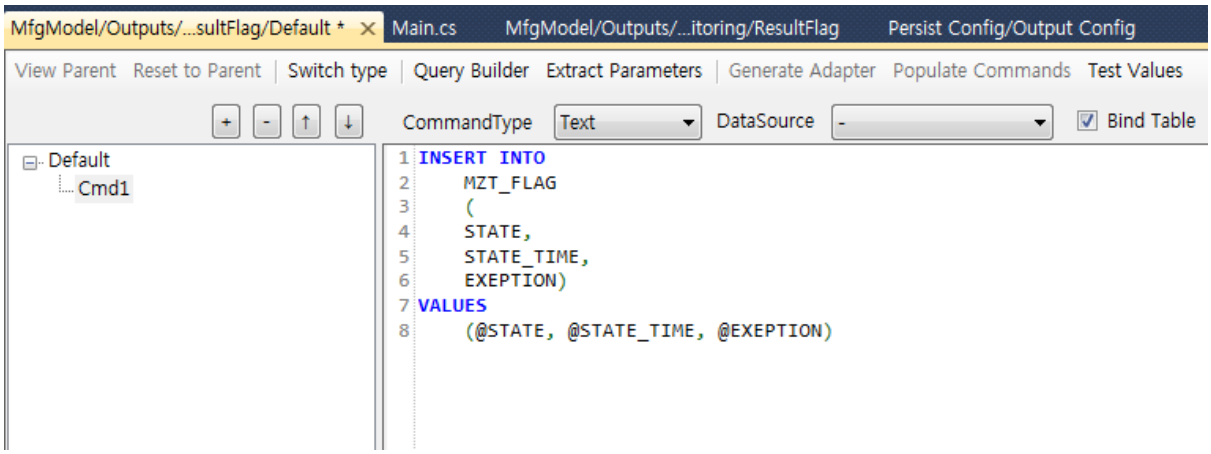
1. Output 결과 기록을 위한 Output DataItem 을 정의합니다.



2. output persist config 에 Monitoring Table 속성에 결과 기록을 위한 Output DataItem ('1'에서 정의한 Item) 을 지정합니다.
3. Main Control 의 ShutDown Action 을 구현합니다.

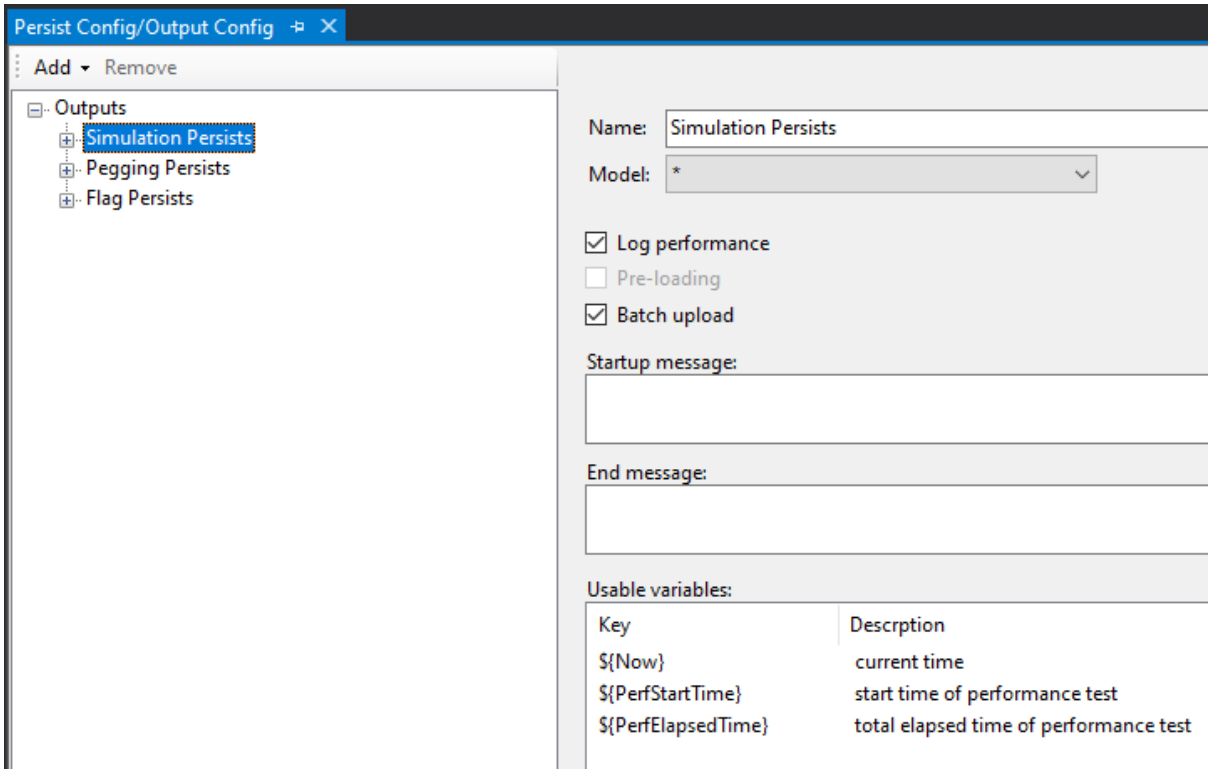
```
public void SHUTDOWN_0(ModelTask task, ref bool handled)
{
    ResultFlag result = new ResultFlag();
    result.STATE_TIME = DateTime.Now;
    result.STATE = task.HasErrors ? "FAIL" : "SUCCESS";
    result.EXCEPTION = task.Exception.Message;

    OutputMart.Instance.ResultFlag.Add(result);
}
```

4. 결과 Output 에 대한 DataAction 설정을 통해 Monitoring 용 DB 에 결과를 기록합니다.



Output Data Group 저장 옵션 설정

Data Group 별 저장 옵션을 정의할 수 있습니다. Tree 메뉴에서 Data Group 을 선택하면 우측의 속성창을 통해 속성설정이 가능합니다.



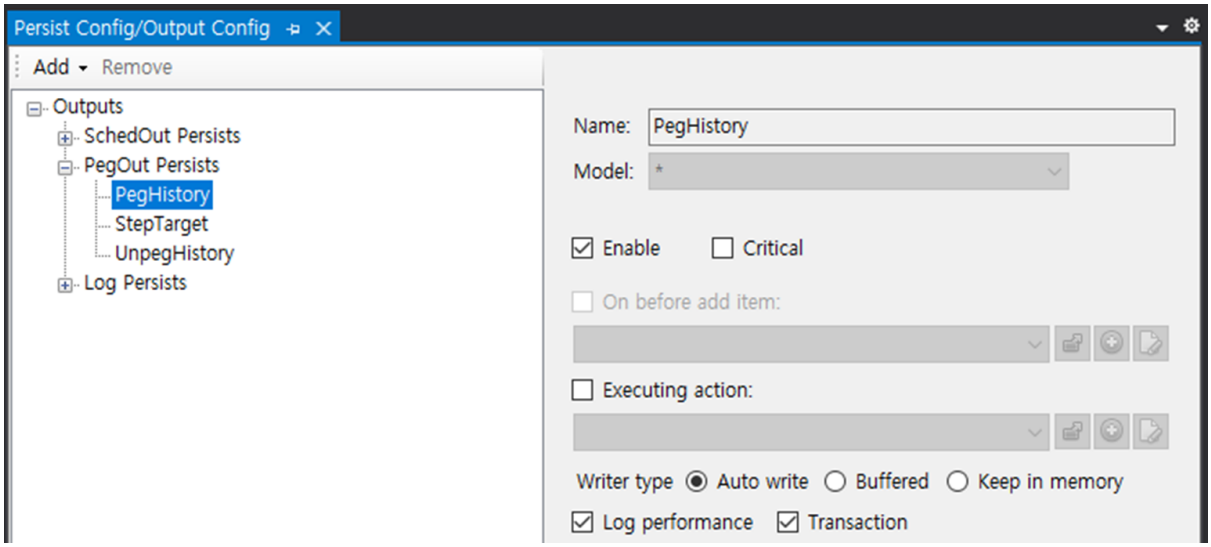
- **Name** : 로딩 시 사용할 Group 명입니다.
- **Log performance** : 전체 Data 저장 소요시간을 기록할지 여부를 설정합니다.
- **Batch upload**: Thread를 할당해서 다른 Group의 테이블과 동시에 DB Save를 수행할지 여부입니다. 예를 들어 Data Group A, B에 Batch upload를 체크하고 Thread Count가 2면, DB Save를 수행할 때 A Group의 테이블 하나와 B Group의 테이블 하나가 동시에 DB에 Write 합니다.
- **Startup message**: Group 에 대한 저장 시작 시점에 Log 에 남길 메시지를 편집합니다. 메시지 편집 시 하단의 Usable variables 를 사용할 수 있습니다.
- **End message**: Group 에 대한 저장 완료 시점에 남길 메시지를 편집합니다. 메시지 편집 시 하단의 Usable variables 를 사용할 수 있습니다.

Usable variable 사용예시

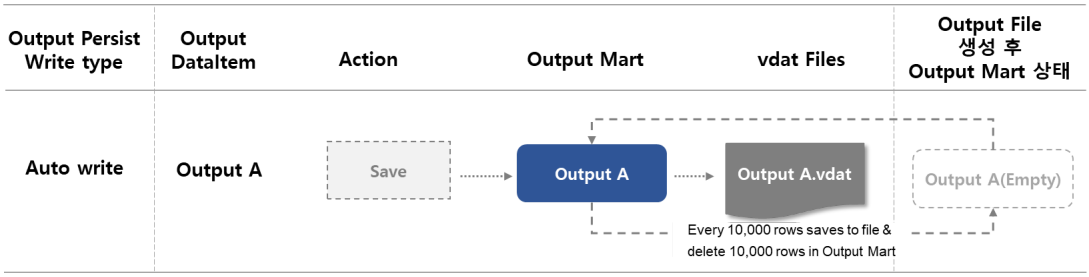
- **Log Message 입력** : This is test log start message at ${Now}
- **결과 출력** : This is test log start message at 2014-09-22 06:05:11

Output DataItem 저장 옵션 설정

개별 DataItem 별로 저장 옵션을 정의할 수 있습니다.

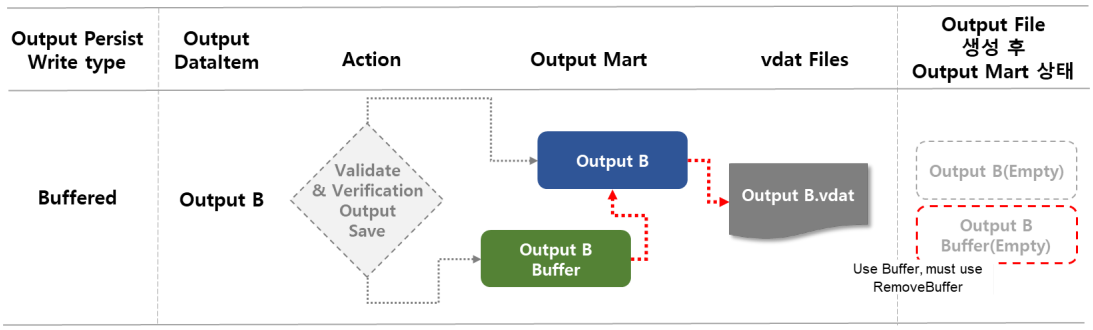


- **Name** : DataItem 명으로 변경이 불가합니다.
- **Enable**: Data 저장 여부를 설정합니다. 옵션을 체크하지 않은 경우 DataItem 이 파일로 저장되지 않습니다.
- **Executing action**: DataItem 의 Data 가 저장 된 후, 추가적인 Data 후처리를 하는 경우 사용됩니다. 해당 Output Data Table 의 Data 를 가공하여 별도의 Outputs 을 추가하는 등의 사용자 코드를 작성할 수 있습니다. 편집 방법은 Input Persist Config의 Action 메뉴 사용방법 과 같습니다.
- **Writer Type** : Outputs 의 저장방식을 설정합니다. 아래는 각 옵션별 저장 형태입니다.
  - **Auto write** : 기본 옵션, OutputMart 의 Data 가 1만 건 마다 Output File 로 Data 를 쓰고, 파일로 저장된 Data 는 Memory 상에서 삭제합니다.

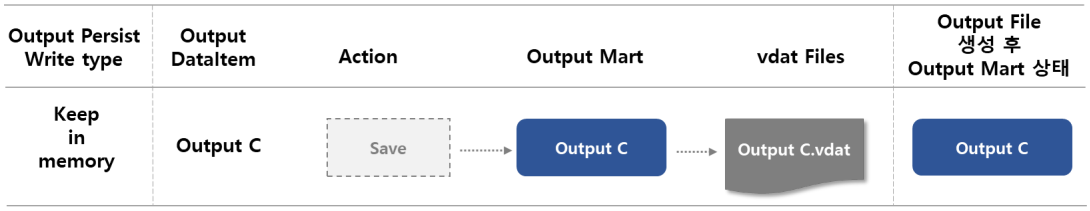


  - **Buffered** : Output 정의 시 Buffer 저장소를 하나 더 만듭니다. OutputMart 에 데이터가 저장 된 후에는 속성 값을 변경 할 수 없기 때문에, 버퍼에 결과 정보를 넣은 후, 결과의 속성 값을 변경하여 최종 OutputMart 에 저장하기 위한 용도입니다. Output Data 의 속성이 모두 확정되지 않은 경우 임시로 Data 를 Buffer 에 저장해 두었다가 속성이 확정되는 시점에 Output 으로 옮기고 Buffer 에서 삭제합니다. 시뮬레이션 종료시점에 Buffer 에 남아있는 Data 는 Output Data 로 저장되기 때문에 Buffer 에 대한 관리코드는 사용자가 직접 작성 해야 합니다. Output 으로 저장된 Data 는 Auto

write 와 같이 1만개 단위로 Output file 에 저장한 후 Memory 상에서 삭제됩니다. 아래 Buffered 로 설정된 Output 에 Data 를 기록하는 **예제코드**를 참조합니다.



- **Keep in memory** : 전체 작업의 시작부터 종료시점까지 Memory 상에 유지되며, 중간에 Output File로 기록하지 않습니다.



- **Transaction** : DataItem에 등록된 DataAction에서 등록된 Command 수행 시 Commit 시점을 지정하기 위한 속성입니다.
  - 예제. 하나의 DataAction에 Cmd1, Cmd2, Cmd3 이 존재할 때
    - True : Cmd1 → Cmd2 → Cmd3 → Commit 수행
    - False : Cmd1 → Commit 수행 → Cmd2 → Commit 수행 → Cmd3 → Commit 수행

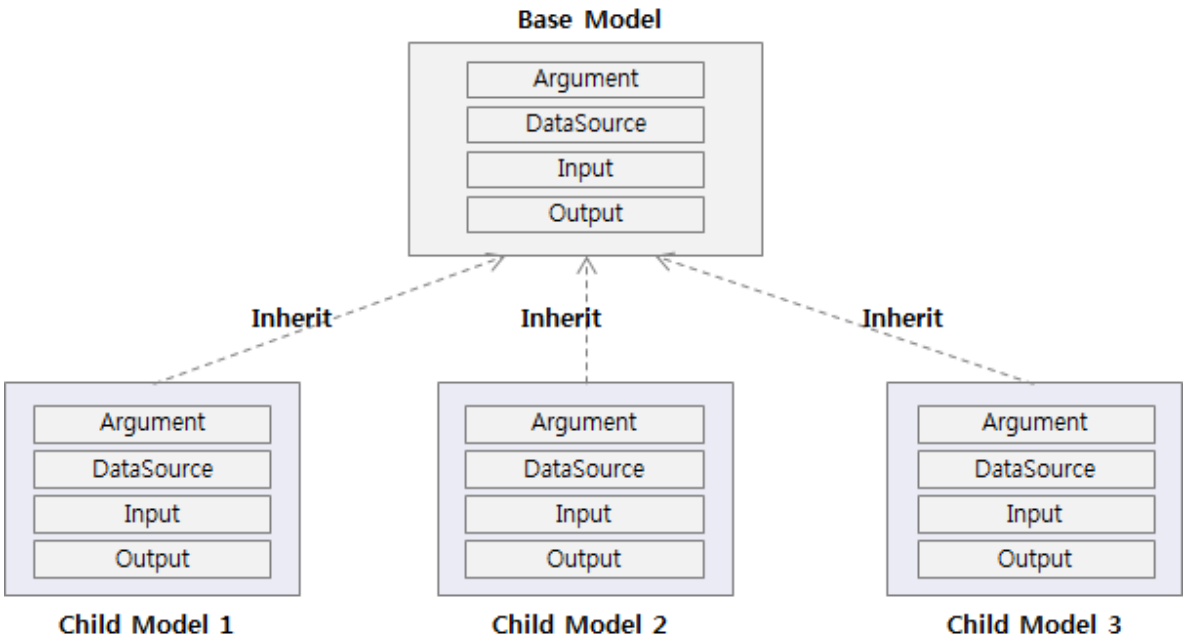
Output DataItem 저장 옵션 설정(Buffered)

```
private void WriteEqpLoadingPlan(ILot lot, IHandlingBatch hb, AoEquipment eqp)
{
    SiteLot mLot = lot as SiteLot;
    string eqpID = eqp.EqpID;

    // Buffer 에 저장된 작업 시작 정보를 찾음
    // FindBuffer() 사용할 때 Output DataItem의 Schema의 key 값으로 Buffer를 찾음
    var plan = OutputMart.Instance.EqpLoadingPlan.FindBuffer(eqpID, mLot.LotID, mLot.Cur
rentStep.StepID);

    if (plan ==null)
    {
```

```
        plan = new EqpLoadingPlan();
        plan.EQP_ID = eqpID;
        plan.LOT_ID = mLot.LotID;
        plan.STEP_ID = mLot.CurrentStep.StepID;
        plan.START_TIME = mLot.CurrentPlan.StartTime;
        plan.PRODUCT_ID = mLot.Product.ProductID;
        plan.PROCESS_ID = mLot.Process.ProcessID;

        // 작업 시작정보가 없는 경우 Buffer 에 추가
        OutputMart.Instance.EqpLoadingPlan.AddBuffer(plan);
    }
    else
    {
        // 시작 정보가 있는 경우 종료 정보를 쓰고 최종 Output 에 추가함
        // Buffer 의 Data 를 삭제
        OutputMart.Instance.EqpLoadingPlan.RemoveBuffer(plan);
        plan.END_TIME = eqp.NowDT;
        plan.QTY = mLot.UnitQty;

        // Output Data 에 추가
        OutputMart.Instance.EqpLoadingPlan.Add(plan);
    }
}
```

다중 Model 프로젝트

MOZART Project 를 구성하는 요소 중 Model 은 복수 개를 생성할 수 있습니다. 단 첫 번째로 추가된 Model 이 Base Model 이 되며, 이후 추가되는 Model 은 Base Model 을 상속받아 생성됩니다. 아래 그림은 Base Model 과 Child Model 의 관계를 도식화 합니다.



Scheduling 시스템을 라인별로 구축하게 되는 경우 기본적인 Inputs, Outputs 에 대한 구조가 거의 유사하고 Data Source 나 Arguments 등 일부만 차이가 나게 됩니다. 이런 경우에 초기 구축 라인의 Model 은 Base Model 로 하고 이후 Model 을 상속을 통해 만들 수 있습니다. 상속을 받은 Child Model 의 경우 Base Model 과의 관계를 유지하기 위해 편집에 제약이 발생합니다. 다음은 Model 을 구성하는 항목별 편집 제약에 대해 설명합니다.

Child Model의 편집 제약사항

제약에 대한 대상은 모두 Base Model 의 항목을 대상으로 합니다. Base Model 의 항목에 대한 삭제는 기본적으로 수행이 불가하며, 항목별 신규 Item 의 추가는 가능합니다. 다만 수정이 필요한 경우, 항목에 따라 허용하는 범위가 다르게 설정되어 있습니다. 항목별 상세 내용은 아래와 같습니다.

Arguments : Input Arguments 와 Config Arguments 에 대한 편집 제약입니다.

- Arguments 추가 : O
- Arguments 수정 : X
- Arguments 삭제 : X
- Config Arguments 의 값 Mapping Data : X

Database

- Database 추가 : O
- Database 명 변경 : X

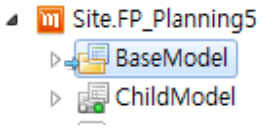
- Database 연결정보 변경 : O
- Database 연결정보 추가 및 순서 조정: O
- Database 삭제 : X

Inputs, Outputs

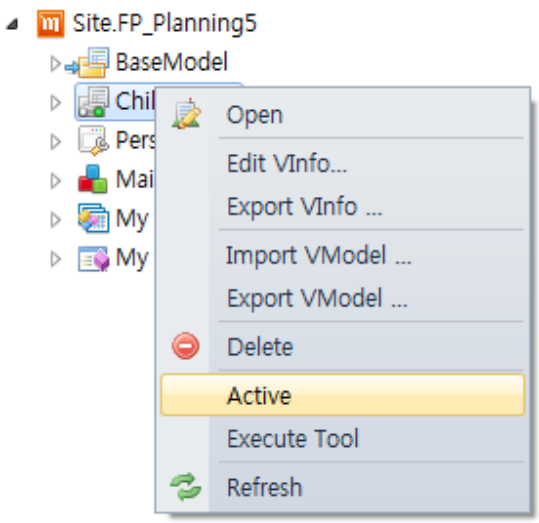
- Schema 추가 : O
- Schema 수정 : X (속성추가, 변경)
- Schema 삭제 : X
- DataAction 추가 : O
- DataAction 수정 : O (DataAction 이름 변경 외 Query 등 모든 부분 수정 가능)
- DataAction 이름변경 : X
- DataAction 삭제 : X

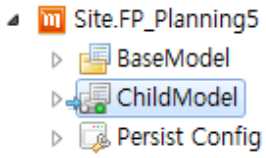
Child Model의 추가 및 활성화

Child Model 은 Model 추가 방식과 동일하게 MOZART Project 의 최상위 노드에서 **[Add > VModel]** 메뉴를 실행하여 추가합니다. Model 명칭은 Base Model 과 달라야 하며, 아래 그림과 같이 Base Model 과 Child Model 의 아이콘이 다릅니다.

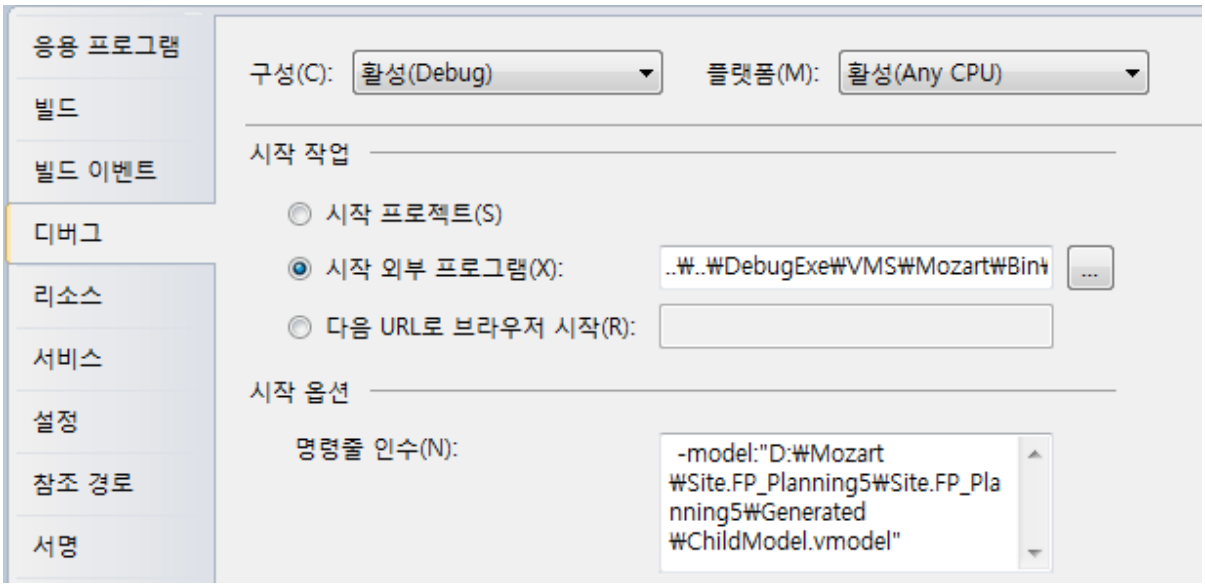


상기 Base Model 표시 아이콘에 파란색 화살표는 현재 활성화된 Model 을 의미합니다. 활성화된 Model 은 Debugging 시에 Project 에서 사용하는 Model 입니다. 사용자가 Debugging 시점에 사용할 Model 을 설정하기 위해서는 활성화 대상 Model 을 선택한 후 팝업메뉴에서 **[Active]** 메뉴를 실행합니다. 실행하면 대상 Model 의 아이콘에 파란색 화살표로 활성화된 상태를 표시합니다.

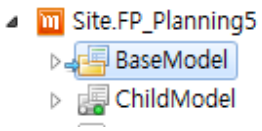




활성화된 Model 정보는 Project 속성 정보에 실행 Model 로 설정됩니다. 아래 그림의 명령줄 인수에서 Model 이 활성화된 Model 로 자동 설정됩니다.



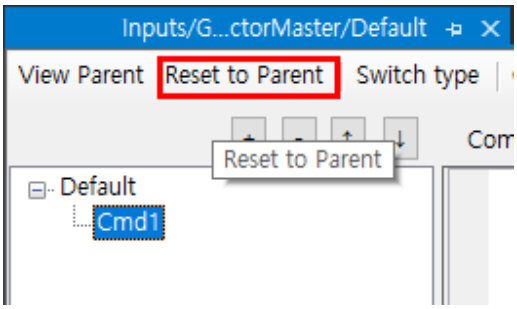
초록색 점의 의미



초록색 점

Base Model 과 Child Model 이 있을 때 Child Model 은 Base Model 을 참조하여 생성되어 있습니다. 만약 Child Model 이 수정되면 재정의(Override) 되었다는 표시로 초록색 점이 생성되게 됩니다.

DataAction 의 경우 아래 **Reset to Parent** 기능을 통해 다시 Base Model 의 값으로 복원이 가능합니다.

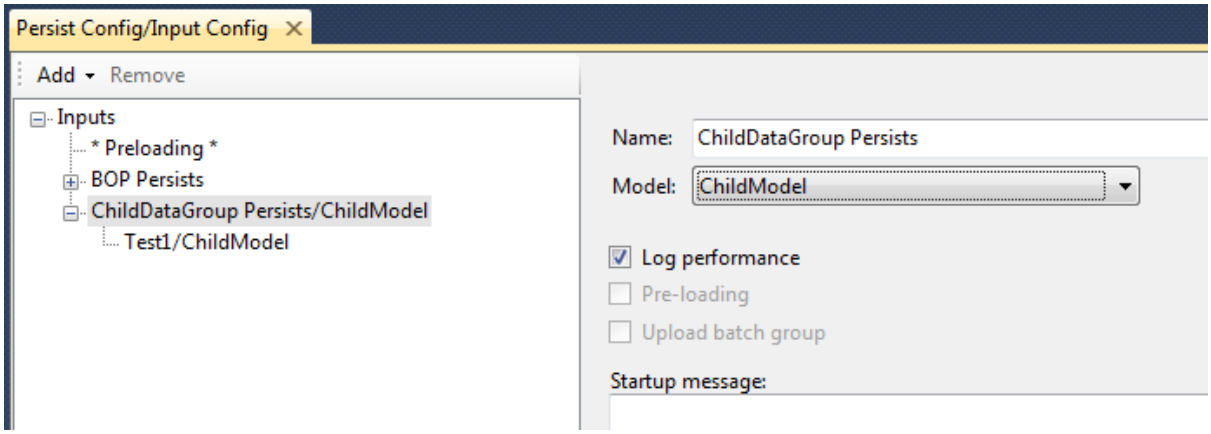


Model 별 Logic 작성

시뮬레이션 Logic 개발 시에 Model 별로 다른 Logic 을 적용해야 하는 경우 혹은 특정 Model 의 Arguments 를 사용하여 Logic 을 개발할 경우에 Model 을 참조하거나 설정하는 방식을 설명합니다.

Persist Config 에서 Model 별 Logic 구성 방법

Persist Group 별로 Model 을 지정할 수 있습니다. 아래 그림과 같이 Model 설정 콤보박스에서 Model 을 선택하면 해당 Model 에서만 Data 를 로딩하게 됩니다. 개별 Data 를 로드할 때 Model 별로 처리하기 위해서는 하단의 Logic 구현 시 Model 확인 방법을 통해 사용자가 Model 을 구분해서 구현합니다.



Logic 구현시 Model 을 확인하는 방법

Model Context 객체의 Vmodel 명 속성을 사용하여 현재 실행 중인 Model 을 확인합니다. VModel 명은 Tree 에 표시된 Model 표시명과 같습니다. 아래는 사용 예제입니다.

```
if (ModelContext.Current.VModelName == "ChildModel")
{
    // Child Model 에 대한 처리 Logic 구현
    // 예: 특정 Model 에 대해서만 Data 를 로딩 혹은 로딩 제외
}
```

Project 에 추가된 모든 Model 파일은 **[Project Root Folder /Generated]** 폴더에 Model 명.vmodel 파일로 생성됩니다. 배포 시에는 Base Model 과 Child Model 의 배포방식에 차이가 없습니다.

MY OBJECTS

My Objects 개요

MOZART 는 대량 Data 의 빠른 처리를 목표로 함에 따라 필요한 Data 를 Logic 처리에 용이한 형태의 Data Model 로 변환하여 Memory 에 로드한 상태에서 Logic 처리를 시작합니다. Logic 처리를 담당하는 Main Module 과 Execution Module 의 실행 시 사용하게 될 Data

Model 클래스를 새로 정의하거나, Library 에서 제공하는 Data Model 을 재정의하여 사용하기 위한 기능입니다.

My Object 의 기능 구성

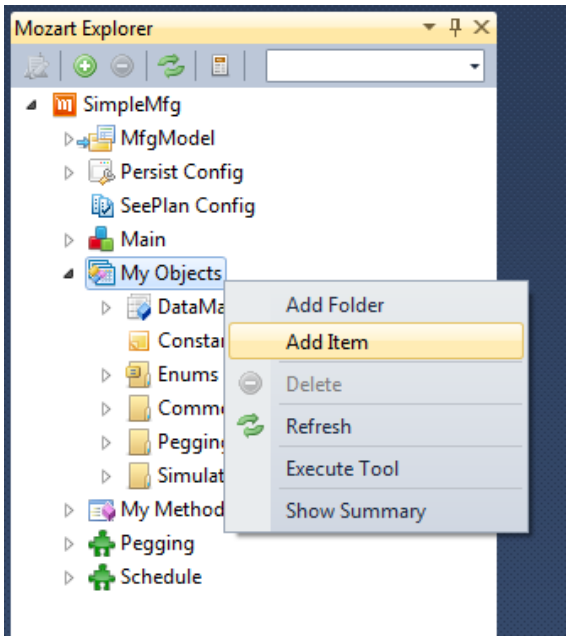
- **Library 구동을 위한 필수 Data 정의**
  Pegging 및 Simulation(Schedule) 수행을 위해 필수적인 Data Model 클래스 와 Interface Data 에 대한 자동 생성 및 수정 기능입니다. 사용자는 사용자 조직의 실제 Data 관리상 필요한 Data 를 추가하여 사용하는 것이 가능합니다. 사용자 조직의 Logic 을 개발할 때에는 새로 정의된 Data 클래스를 사용하여 Logic 을 구현합니다. 기본적으로 Execution Module(Pegging, Simulation) 을 추가할 때 해당 Module 에 필요한 Data 클래스와 두 가지 Module 에서 공통으로 필요한 Data 클래스를 자동으로 Project 에 추가합니다. 자세한 사항은 ***My Object 기능 사용방법*** 을 참고하시기 바랍니다.
- **Data클래스 정의**
  Input, Output DataItem 으로 정의할 수 있는 단순한 구조의 Data 로 Logic 구성이 용이하지 않을 때, Logic 에서 사용하기 적합한 형태의 복합적인 DMO(Data Manipulation Object) 클래스를 사용자 조직에 맞도록 구성할 때 신규 Data 클래스를 정의할 수 있습니다. 자세한 사항은 ***My Object 기능 사용방법*** 을 참고하시기 바랍니다.
- **Constants 정의**
  사용자 조직의 Logic 을 구성하기 위해 필요한 상수(Constants) 를 정의해서 사용합니다. 자세한 사항은 ***Constant 정의 방법*** 을 참고하시기 바랍니다.
- **Enum 정의**
  Enum 타입의 Data 를 정의하여 사용합니다. 자세한 사항은 ***Enum 정의 방법*** 을 참고하시기 바랍니다.
- **DataMart 정의**
  Model 의 Inputs/Outputs 에 정의한 Data 혹은 View 에 대해서는 자동으로 InputMart, TempMart, OutputMart 에 Collection 이 정의됩니다. 또한 My Objects 에서 정의한 Data 클래스의 Collection 도 InputMart 에 자동으로 생성됩니다. 이렇게 정의된 내용을 확인하고 추가적으로 DataMart 에 사용자 정의 Data Collection 을 정의하기 위해 본 기능을 사용합니다. 자세한 사항은 ***DataMart 사용방법*** 을 참고하시기 바랍니다.

My Objects 사용 방법

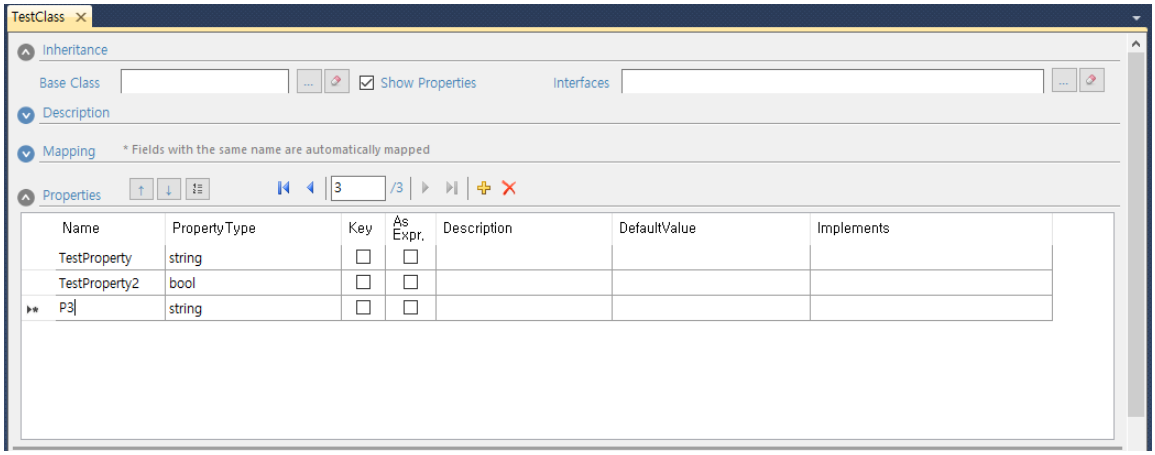
My Objects 는 기본적으로 Data 클래스를 정의하는 기능입니다. 신규로 Data 클래스를 정의하는 방식은 두 가지가 있습니다. Library 에서 기본 제공하는 Data Model 클래스를 상속받아 구현하는 방식과 초기부터 새로 만드는 방법이 있습니다. 본 장에서는 Object 를 정의하는 기능적인 부분을 위주로 설명합니다.

사용자 정의 Data 클래스 추가 및 기본 작성방법

1. My Objects 노드에서 **[Add Folder / Add Item]** 메뉴를 통해 신규 폴더나 Data 클래스를 추가합니다

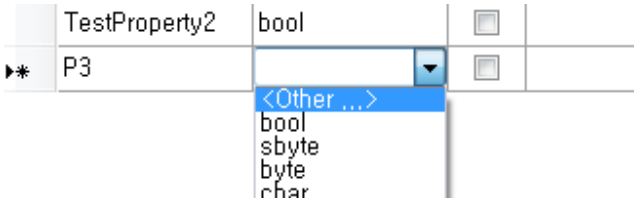


2. Column (Property) 을 입력하고 Type 및 속성정보를 편집합니다.

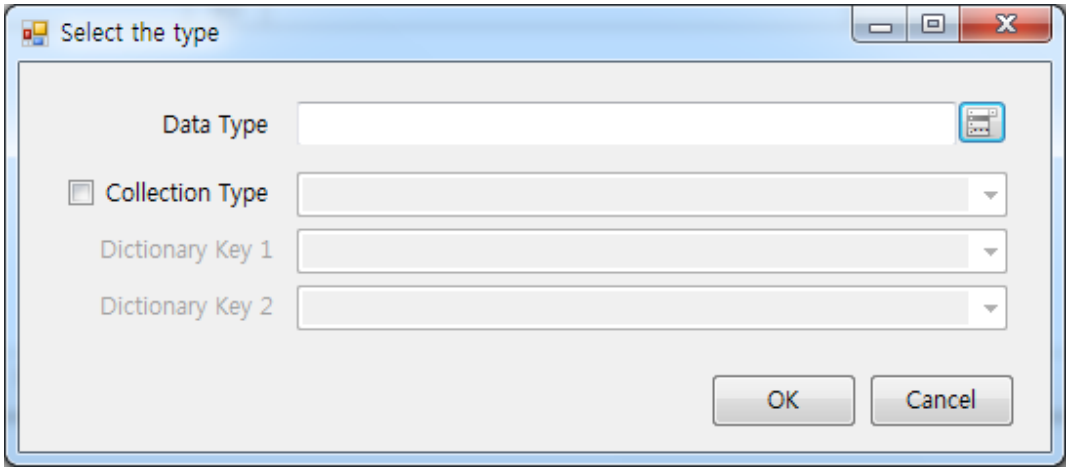


Data 클래스의 Property 에 대한 속성을 정의합니다.

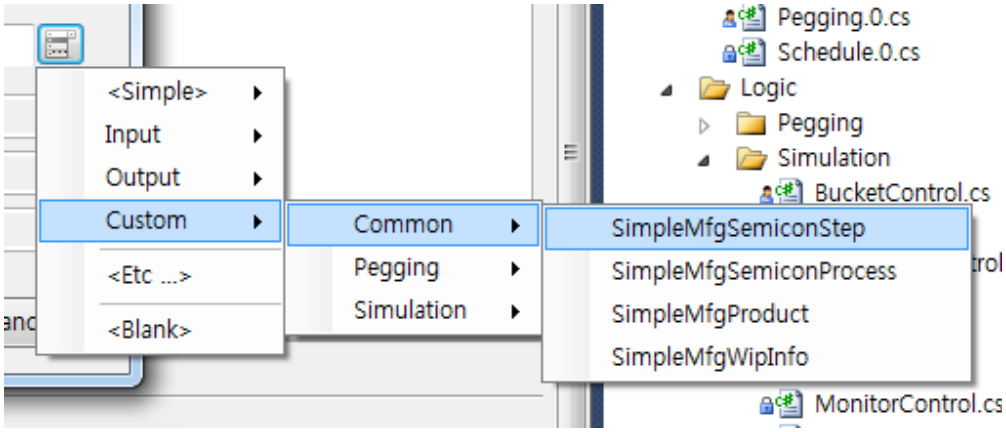
- **Name** : Property 명
- **Property Type** : Property Data 타입입니다. 기본 Data 타입의 경우 콤보박스에서 바로 선택하여 사용할 수 있으며, Project 에서 정의한 Inputs, Outputs, My Objects 의 모든 Data 타입의 경우 콤보박스의 를 통해 선택하여 지정할 수 있습니다.



Data Type 입력창의 오른쪽 버튼을 클릭하여 Inputs / Outputs / My Objects 의 항목을 선택할 수 있습니다.

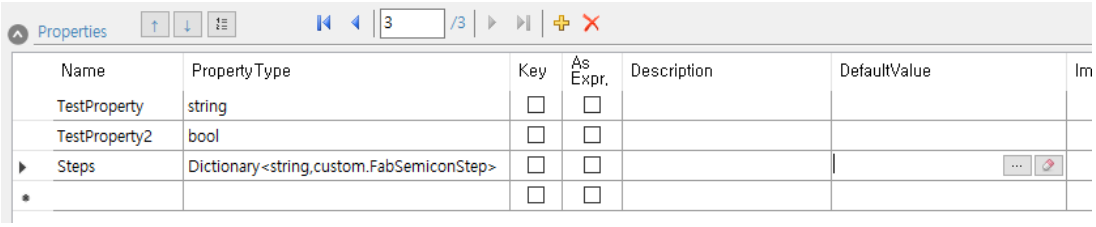


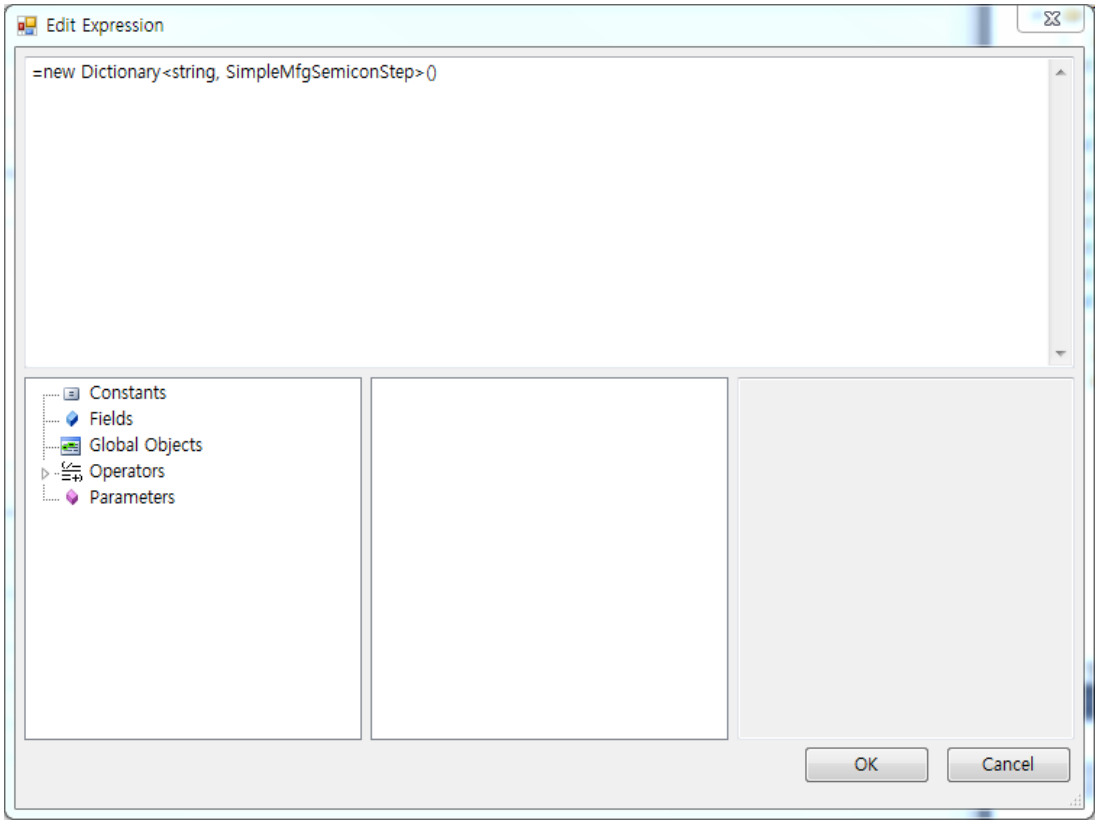
<Etc…> 를 통해 시스템 Library 에서 제공하는 모든 Data 타입을 사용하여 지정하는 것도 가능합니다.



Collection Type 속성의 경우, 일반적으로 초기화가 필요합니다. DefaultValue 편집을 통해 초기값 및 초기값을 정의하는 Experession 을 입력합니다.

- **Key** : Property 의 Key 여부
- **As Expr**. : As Default Expression의 약자로 Property 를 고정 상수(Constants) 처럼 사용하고자 하는 경우 설정합니다. 이때는 **DefaultValue**를 표현식으로 구현해야 합니다.
- **Description** : Property 설명
- **Implements** : Data 클래스가 Interface 의 구현으로 정의되거나 혹은 추상클래스로부터 상속받아 생성되는 경우, 필수 구현이 필요한 속성을 Mapping 하는 부분입니다. Interface 의 경우 이를 설정하여 명시적 구현을 할 수 있습니다.
- **DefaultValue** : 기본 값 입니다. Property Type 을 Collection 형태로 정의하거나 시스템에서 정의한 다른 객체를 속성으로 정의하는 경우 Default 를 지정해주어야 합니다. 값은 Expression 형태로 입력할 수 있습니다. 값을 입력하기 위해서는 Cell 우측의 **[...]** 버튼을 클릭하여 Expression 편집창을 활성화합니다. 최 우측의 **[지우개]** 버튼은 Expression 을 삭제하는 데 사용합니다.

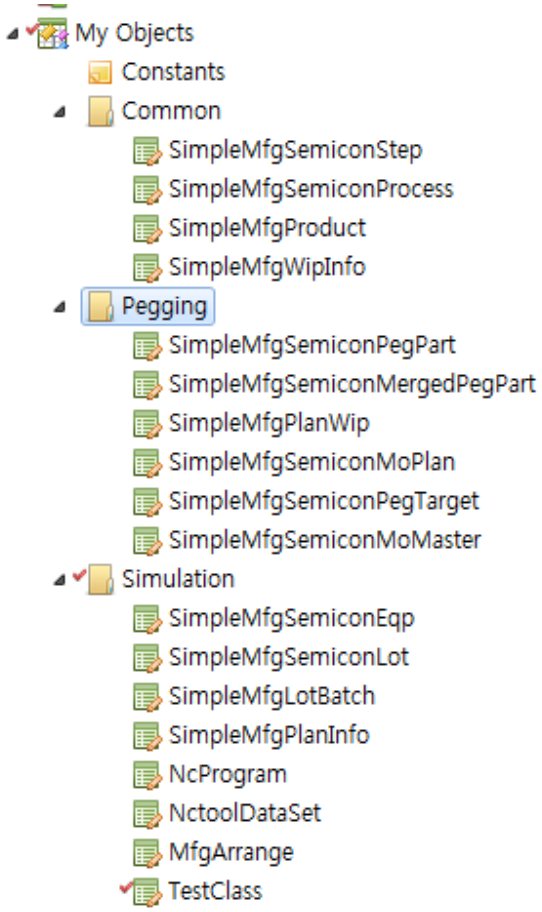




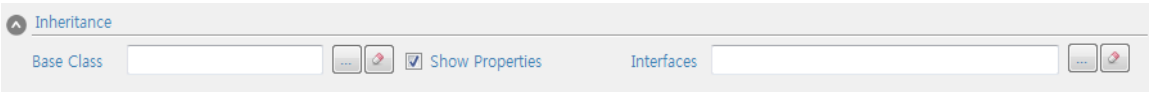
3. 모든 편집을 마치고 저장합니다.

Library 클래스로부터 Data 클래스 정의방법

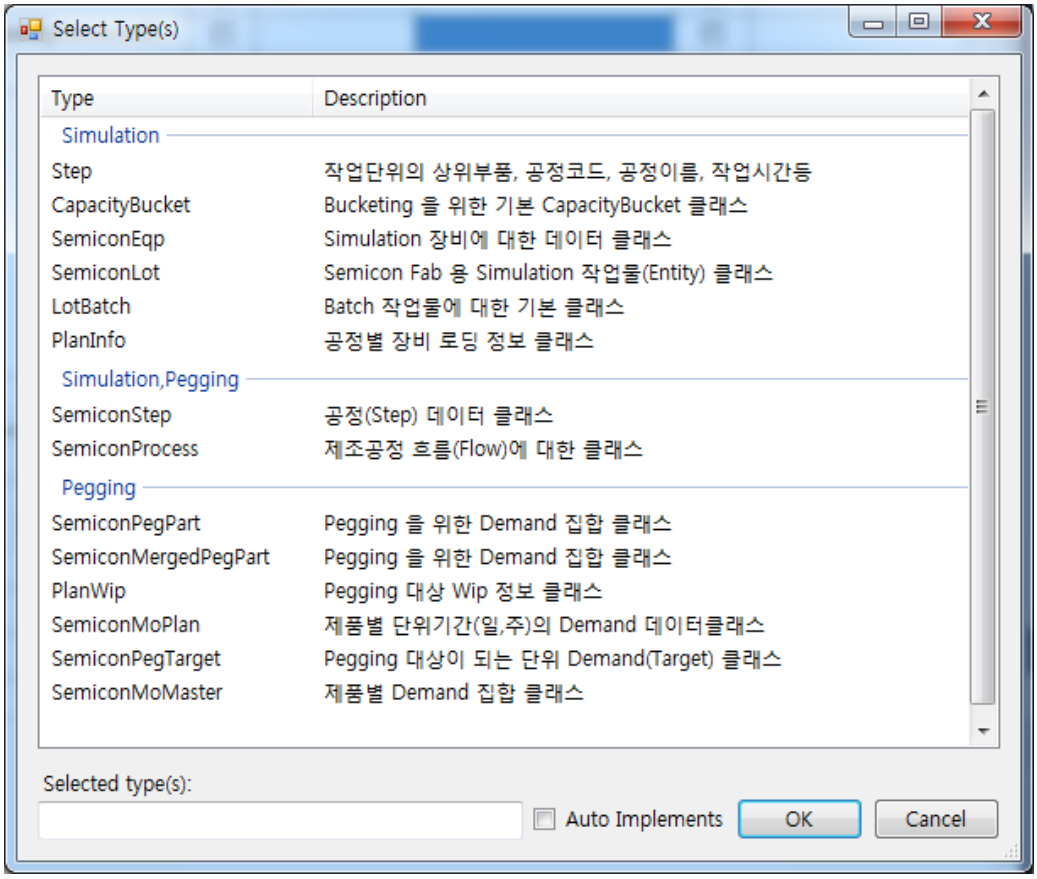
1. Execution Module 추가 시 필수 구현이 필요한 Data 클래스의 경우, 자동으로 추가됩니다. Simulation Module 관련 Data 는 Simulation 폴더 하위에 추가되며, Pegging 은 Pegging 폴더 하위에, 그리고 두 가지 Module 을 구현하기 위한 기본 Data 의 경우 Common 폴더에 자동으로 생성됩니다.



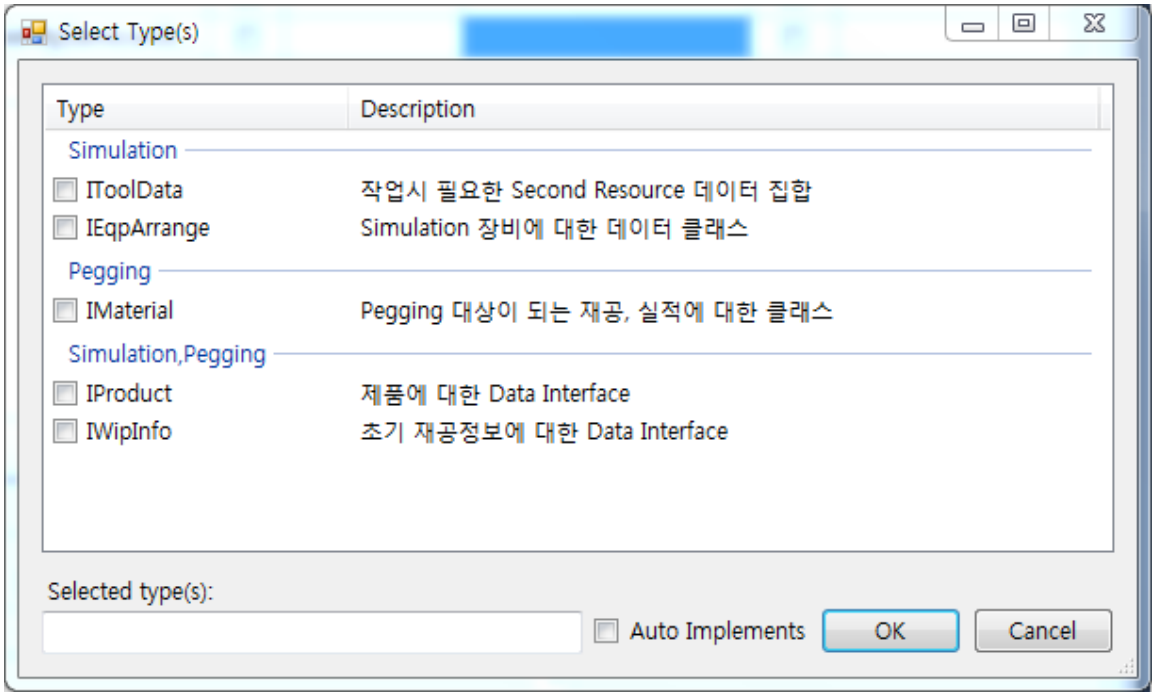
2. 그 외에 Data 를 Library 로부터 추가로 작성하고자 하는 경우에는 Data 클래스 편집창 상단의 Inheritance Section 에서 Base Class 혹은 Interface 를 설정합니다.

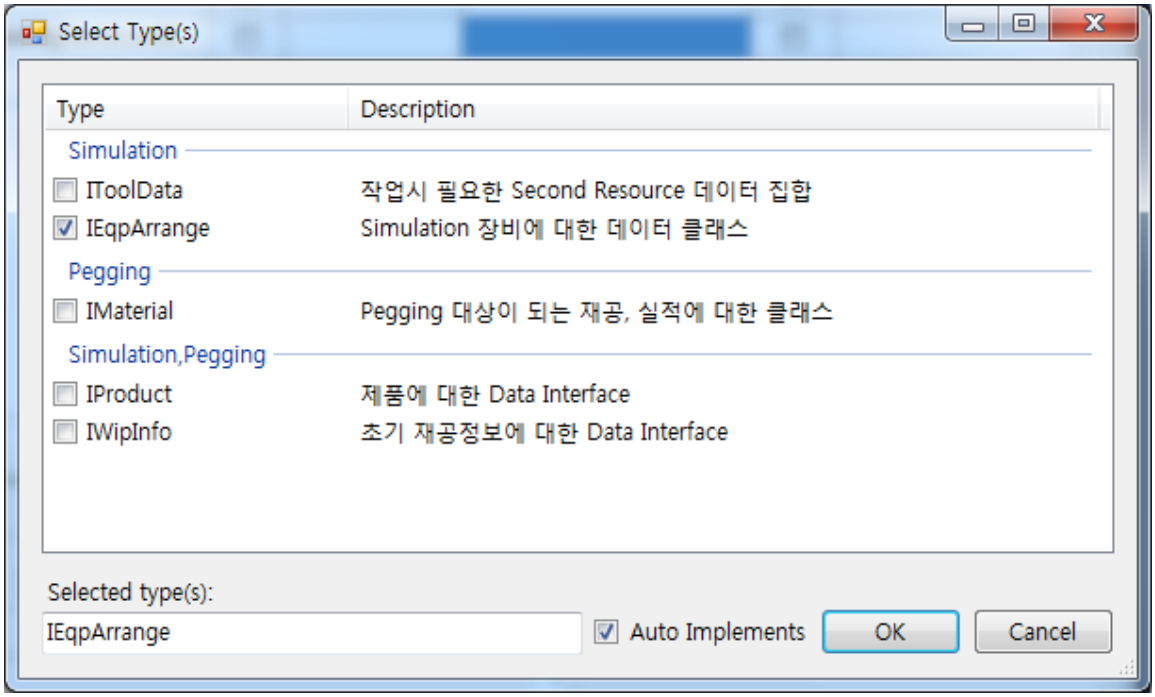


3. Base Class 는 Library 상의 Data 중 Pegging, Simulation 의 주요 Data Model 객체를 표시합니다. […] 버튼을 눌러 Library 클래스 선택창을 열고 상속받기 위한 클래스를 선택합니다. 기본적인 클래스들은 Module 을 선택한 경우, 기본적으로 추가되어 있을 것입니다. 그 외 필요한 클래스를 선택하고 확인하면 해당 클래스를 상속받아 클래스를 정의할 수 있습니다. 하단의 Auto Implements 를 체크한 상태로 저장하면 Base Class 의 속성 명과 동일하게 필수 구현 항목이 자동으로 클래스에 속성으로 추가됩니다.

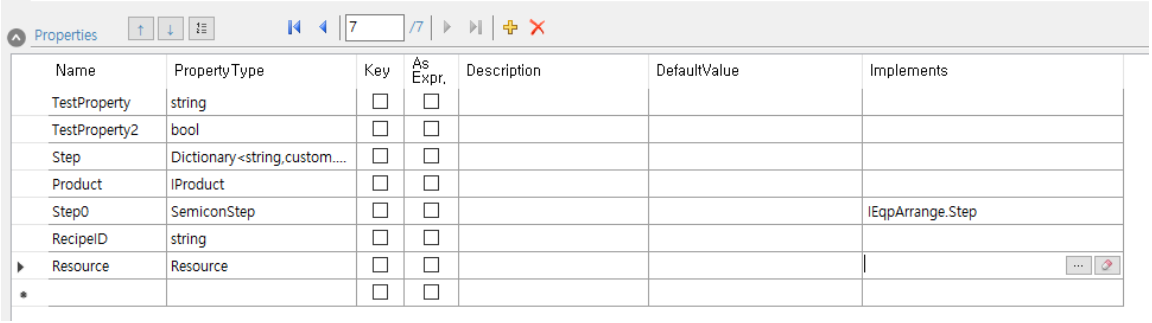


4. Interface 에 대한 상속도 Base Class 와 동일한 방식으로 구현대상이 되는 Interface 를 조회하고 선택하여 구현할 수 있습니다. 역시 Auto Implements 를 체크하는 경우, Interface 명과 동일하게 속성들이 자동으로 생성됩니다.

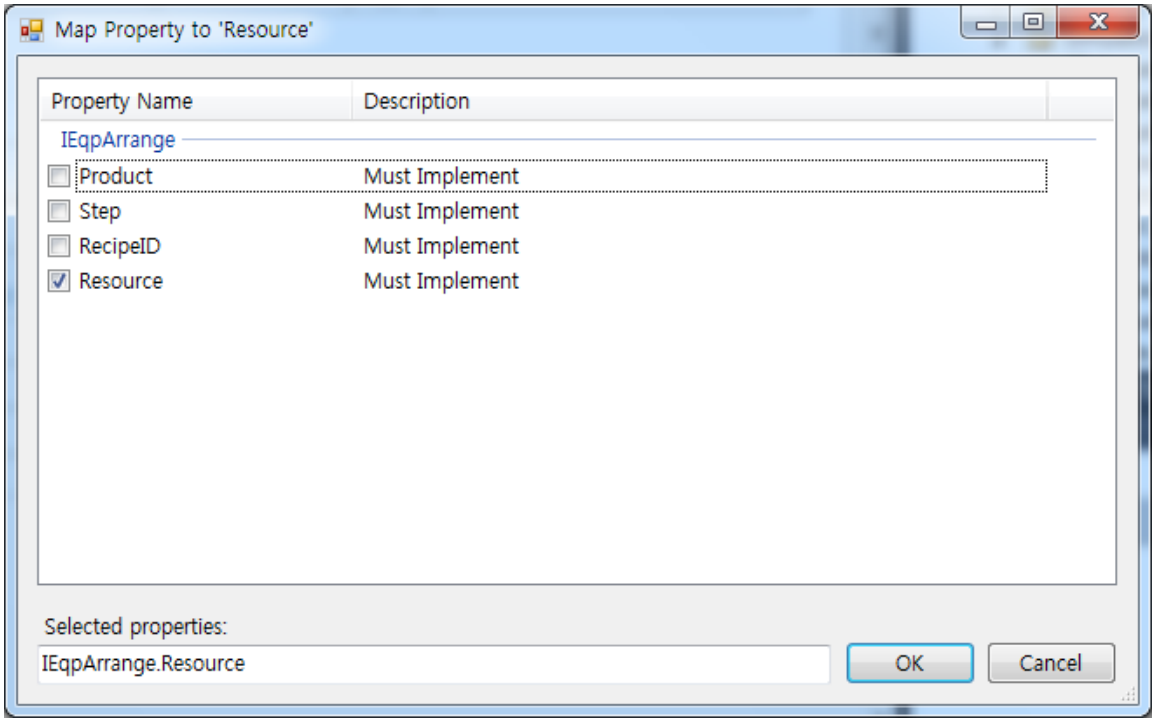


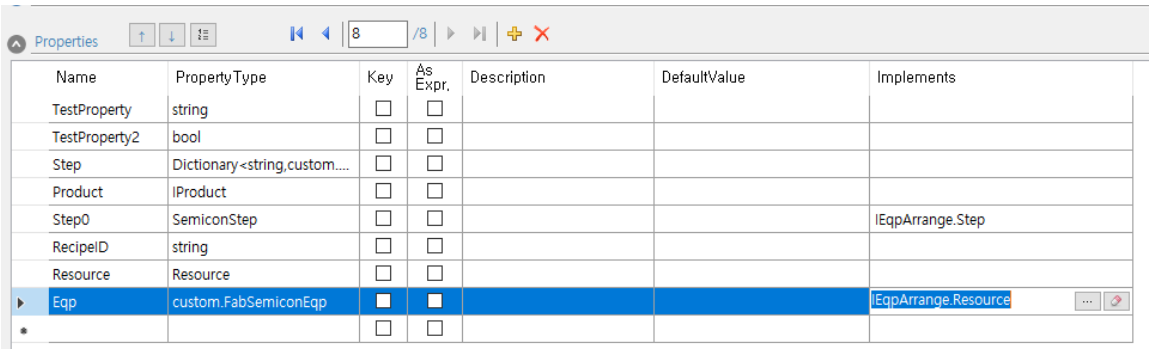


상기 그림과 같이 설정한 경우, 아래와 같이 자동으로 필드가 채워집니다. 하위의 Product, Step, RecipeID, Resource 가 IEqpArrange Interface 를 구현했을 때 자동으로 생성되는 속성입니다. 기본적으로는 동일한 명칭으로 구현됩니다.



이때, Resource 항목을 사용자 사이트의 특정 클래스 Type 으로 정의하고 Interface.Resource 를 명시적으로 구현하기 위해서는 Implements 항목을 선택하면 됩니다. 아래 그림을 참고하시기 바랍니다. 명시적으로 구현할 속성의 Implements Column 에 표시되는 버튼을 클릭하여 Interface 의 명시적 구현 속성을 선택합니다.





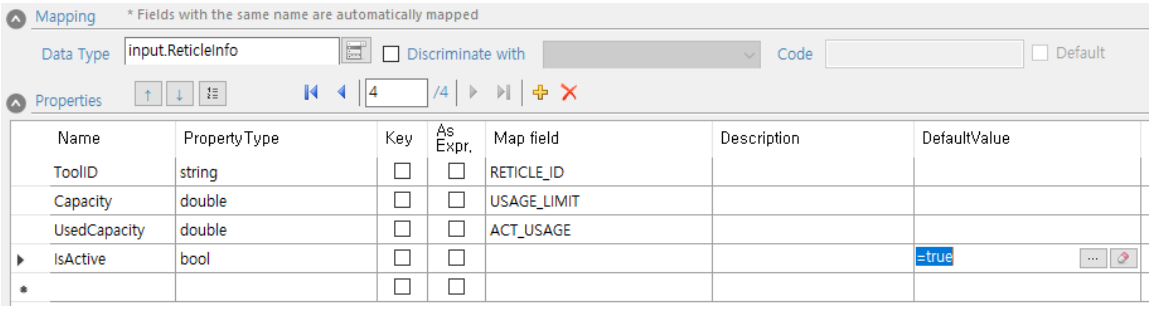
상기와 같이 설정한 경우, 자동 생성되는 소스는 아래와 같이 Interface 에 대해 명시적으로 구현하는 소스가 생성됩니다.

```
[System.SerializableAttribute()]
public partial class TestClass : Mozart.SePlan.Semicon.DataModel.IEqpArrange
{
    public virtual string TestProperty { get; set; }

    public virtual bool TestProperty2 { get; set; }

    public virtual Dictionary<string,SimpleMfg.DataModel.SimpleMfgSemiconStep> Steps { get; set; }

    public virtual Mozart.SePlan.Semicon.DataModel.IProduct Product { get; set; }

    public virtual Mozart.SePlan.Semicon.DataModel.SemiconStep Step { get; set; }

    public virtual string RecipeID { get; set; }

    Mozart.SePlan.DataModel.Resource Mozart.SePlan.Semicon.DataModel.IEqpArrange.Resource
    {
        get
        {
            return this.Eqp;
        }
    }
    public virtual SimpleMfg.DataModel.SimpleMfgSemiconEqp Eqp { get; set; }
```

Input Data와 My Object의 Mapping 사용

신규로 정의한 My Objects 의 경우 Input Data 와 같이 자동으로 Data 를 로드하지 못합니다. Data 의 구조가 자동으로 채울 수 있는 구조가 아니기 때문입니다. 그러나 일반적으로 My Objects 에 구현되는 Data 클래스는 Mapping 되는 주요 Input Data 가 있는 경우가 많습니다. 이러한 경우, Input Data 의 Column 과 My Object 의 Property 간에 Mapping 을 통해 My Objects 를 생성할 수 있습니다.

1. Mapping 이 가능한 Data 에 대해 Mapping Source 가 되는 Input Data 를 선택합니다.

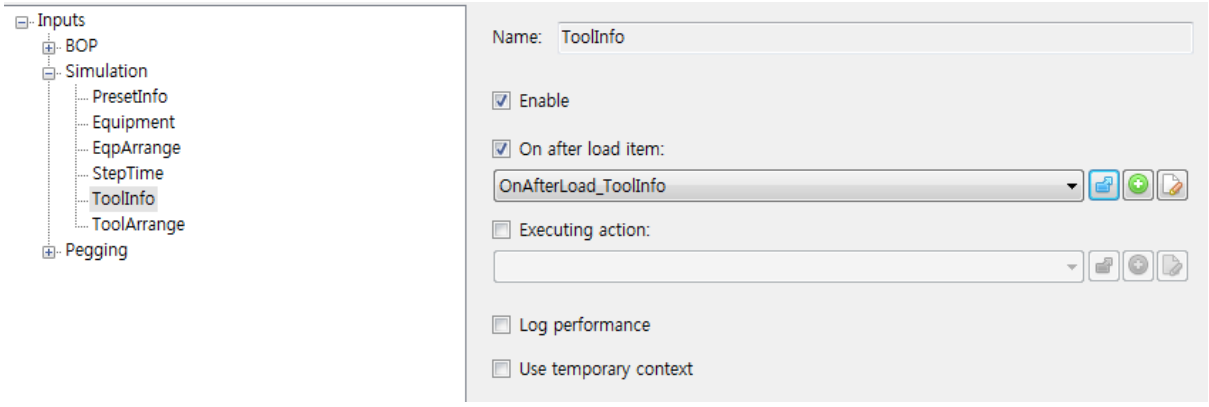


Getting Started

2. Mapping Section 의 Data Type 을 선택하는 경우, Properties 그리드에 “Map field” Column 이 추가되며, 선택한 Data Type 의 속성을 선택할 수 있습니다.

> **Note**
> My Object 의 속성 명과 Mapping Data Type 으로 설정된 Input Data 의 Column 명이 같은 경우, Map field 를 따로 설정하지 않아도 자동으로 Mapping 이 됩니다.

1. Property 와 Mapping 되는 Input Data 의 Column 을 선택하여 Mapping 정보를 생성합니다. 이렇게 만들어진 Mapping 정보는 Input Data 에 “ToMyObject()” 함수를 자동으로 생성하며, 이 함수를 통해 Mapping 이 된 My Object 객체를 생성할 수 있습니다.
2. 이후 Input Persist Config 에서 Mapping 할 Input Data 의 로딩 함수를 통해 Mapping 생성함수를 사용하여 Data 를 로드합니다.



Getting Started

Mapping 함수를 통한 개체 생성 사용 예제

```
public bool OnAfterLoad_ToolInfo(ToolInfo entity)
{
    if (entity.TOOL_STATE != "G")
        return false;
    // Input Data 로부터 Data 클래스 Object 를 생성하며,
    // 이때 Map field 의 값을 자동 변환하여 속성한 상태로 생성합니다.
    NcProgram pgm = entity.ToNcProgram();
    if (InputMart.Instance.NcProgram.ContainsKey(pgm.PgmID) == false)
        InputMart.Instance.NcProgram.Add(pgm.PgmID, pgm);
    return true;
}
```

Mapping 된 My Objects 의 속성과 Map field 간의 Data Type 이 다른 경우, 자동 변환을 지원합니다. 자동 변환 지원 범위는 다음과 같습니다.

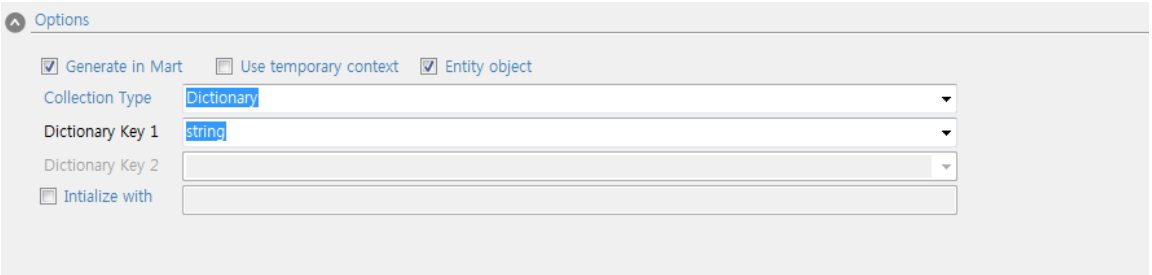
- **String > Number Type** : 문자열 항목을 My Objects 의 Number Type (integer, float, double, decimal, …) 으로 자동 변환합니다. Null 은 허용하지 않습니다.
- **Number Type > String** : 숫자 형식의 값을 문자열로 자동 변환합니다.

- **String : Enum Type** : 문자열 항목을 Enum Type 으로 자동 변환합니다. 아래의 조건을 만족하지 않는 경우, 오류가 발생합니다. 자세한 사항은 **Enum 정의방법**을 참조하시기 바랍니다.
- Library 에서 정의된 Enum 으로 변환하는 경우 : 변환을 위해 Mapping 되는 문자열은 정확히 Enum 의 문자열과 일치하거나, 모두 대문자, 모두 소문자로 구성되어야 합니다.
- 사용자가 **[My Objects > Enums]** 기능을 사용하여 정의한 Enum 으로 변환하는 경우 : My Objects 의 Enums 에서 Map Value 를 설정하지 않은 경우, 기본적으로 Library 에서 제공하는 Enum 으로 변환하는 것과 동일한 방식으로 적용됩니다. 단, Enum 의 Map Value 를 정의한 경우에는 Map Value 와 일치하는 문자열에 대해서만 변환이 가능합니다. 이 때에는 대소문자까지 정확히 일치하도록 문자열을 설정해주어야 합니다.
- **String > DateTime Type** : 날짜 형식의 문자열을 자동으로 DateTime 형식으로 변환합니다. 가능한 날짜 형식은 아래와 같으며, 날짜 시간 조합은 모두 가능합니다.
    - 날짜형식 : yyyy-mm-dd, yyyy/mm/dd
    - 시간형식 : hh24 : mi : ss, [오전 or 오후] hh : mi : ss
    - **DateTime Type > String** : DateTime 형식의 Data 를 문자열로 변환합니다. 변환된 문자열의 형식은 'yyyy-mm-dd [오전 or 오후] hh : mi : ss' 을 따릅니다.

My Object를 Input Mart에 생성하기

Input, Output Data 의 경우, InputMart, OutputMart 에 자동으로 생성됩니다. 반면, My Objects 는 InputMart 에 자동으로 생성되지 않습니다. My Objects 의 경우 기본적으로 Logic 을 개발하기 위해 필요한 Data 로 특정 객체의 속성으로 저장되기도 하며, InputMart 에 별도로 추가시켜 사용할 수 있습니다.

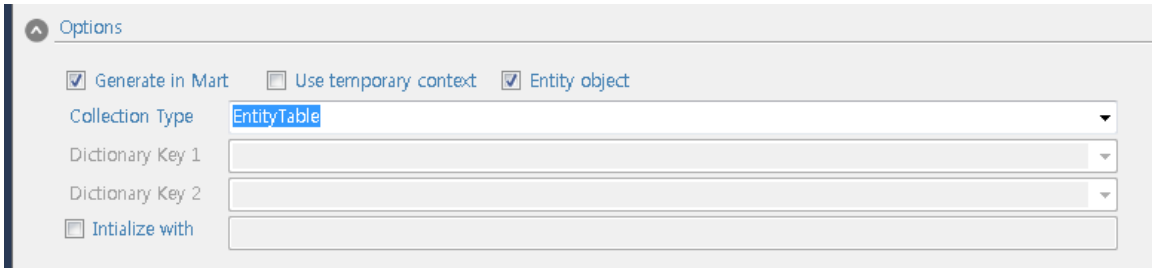
1. My Objects Data 클래스 편집창 하단의 Option Section 을 펼칩니다.

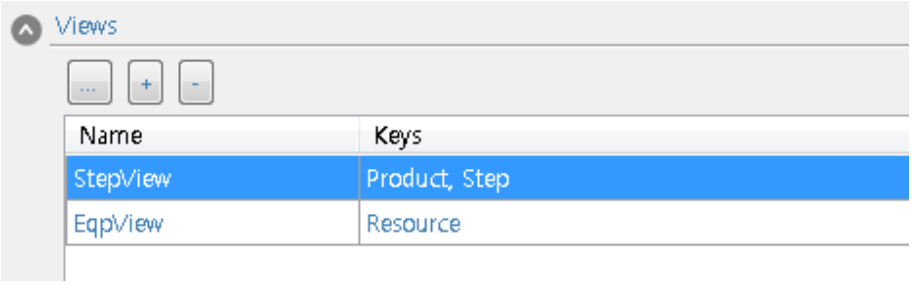


2. Options 에서 InputMart 나 TempMart 에 저장 여부 및 객체의 저장 Collection 형태를 정의할 수 있습니다.
    - **Generate in Mart** : InputMart 에 Data 저장 여부를 결정합니다.

- **Use temporary context** : TempMart 에 Data 를 저장하는 경우에 사용합니다. TempMart 는 Persist 시행동안만 사용하는 DataMart 입니다.
- **Entity object** : Data 를 Entity Object 로 정의할지를 결정합니다. Entity Object 로 정의하는 경우, 저장 시 EntityTable 형태의 저장 구조를 사용할 수 있습니다.
- **Collection Type** : Data 의 저장 Collection 형태를 결정합니다.
- **Dictionary Key1** : Collection Type 을 Dictionary 로 정의하는 경우, Key 에 대한 Data 타입을 설정합니다.
- **Dictionary Key2** : Collection Type 을 Multi Dictionary 로 정의하는 경우, 두 번째 Key 의 Data 타입을 설정합니다.
- **Initialize with** : Collection 의 초기화 코드를 직접 입력합니다.

3. 정의한 Data 구조는 InputMart 를 통해 접근 가능합니다. 아래의 그림은 EntityTable 로 정의한 경우이며, EntityTable 로 Collection Type 을 설정하는 경우에는 하단의 View 를 정의할 수 있습니다. View 정의에 대한 자세한 사항은 Schema Edit > Data View 편집방법을 참조하시기 바랍니다.





InputMart 의 My Object 저장 Data 사용

```
// 1. Data 로드 시점 처리 예
MfgArrange arrange = entity.ToMfgArrange();
arrange.Product = prod;
arrange.Step = step;
arrange.Resource = eqp;
InputMart.Instance.MfgArrange.Rows.Add(arrange);
// 2. Data 사용시점 처리 예
public Ilist<string> GETLOADABLEEQPLIST(DispatchingAgent ad, IHandlingBatch hb, ref
 bool handled, IList<string> prevReturnValue)
```

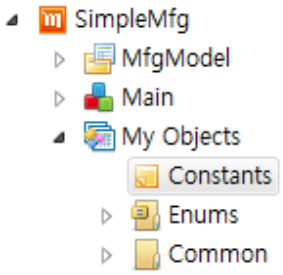
```
{
    SimpleMfgSemiconLot lot = hb.Sample as SimpleMfgSemiconLot;

    var arranges = InputMart.Instance.MfgArrange.Rows.Where(p =>
            (p.Product == lot.Product && p.Step.StepID == lot.CurrentStep.StepID));
    if (arranges == null)
        return prevReturnValue;
    HashSet<string> loadables = new HashSet<string>();
    foreach (MfgArrange arr in arranges)
    {
        loadables.Add(arr.Resource.ResID);
    }
    return loadables.ToList<string>();
}
```

Constants 정의 방법

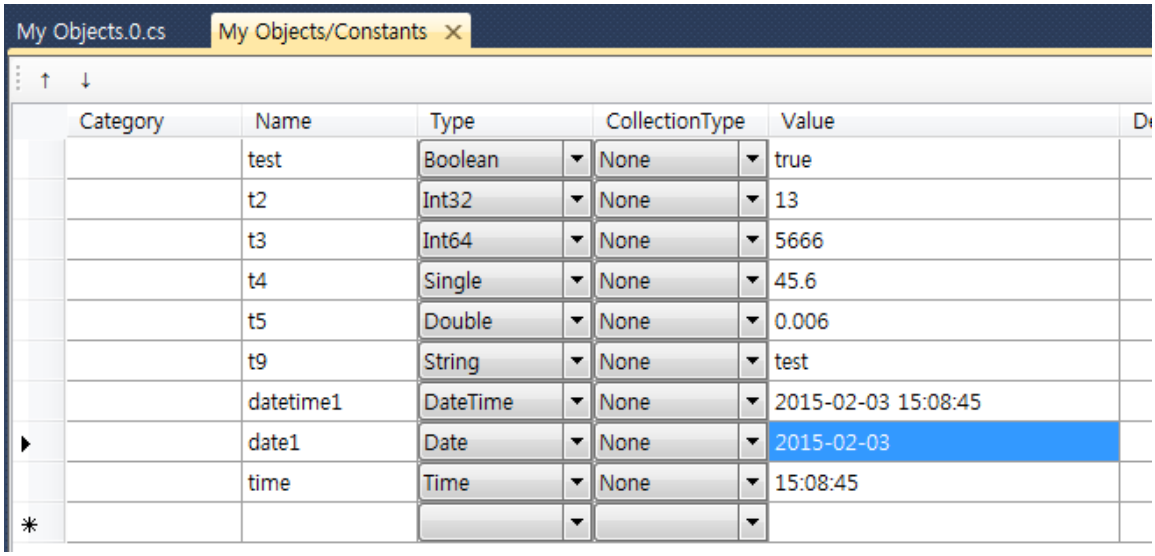
일반적으로 사용자 조직의 Logic 을 개발하는 중에 상수(Constants) 를 정의하는 경우가 많습니다. 이 경우에 My Objects 의 Constants 기능을 통해 상수를 정의해서 사용할 수 있습니다. 아래는 Constants 정의 및 사용방법에 대한 설명입니다.

Constants 정의 및 사용방법

1. [**My Objects > Constants**] 를 클릭하여 Constants 정의 화면을 엽니다.



2. 상수에 대한 정의를 Column 항목에 맞도록 입력합니다.



My Objects.0.cs | My Objects/Constants

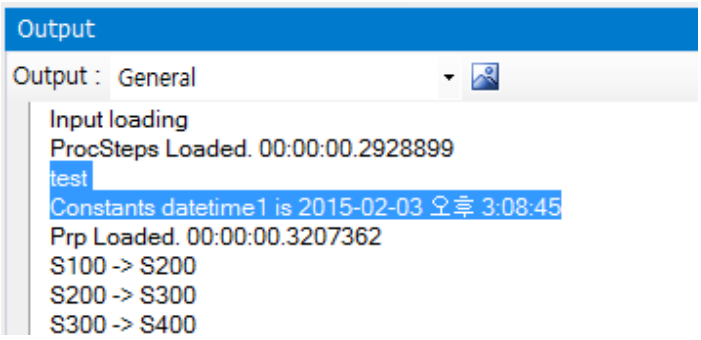
Category	Name	Type	CollectionType	Value
	test	Boolean	None	true
	t2	Int32	None	13
	t3	Int64	None	5666
	t4	Single	None	45.6
	t5	Double	None	0.006
	t9	String	None	test
	datetime1	DateTime	None	2015-02-03 15:08:45
	date1	Date	None	2015-02-03
	time	Time	None	15:08:45

3. 상기 이미지와 같이 Type이 DateTime 인 경우 DateTime 문자열로 상수값을 설정합니다.
4. 사용 시에 Constants 클래스를 사용하여 상수를 참조합니다. 아래는 사용 예시입니다

Constants 값 참조

```
Logger.MonitorInfo(Constants.t9);
Logger.MonitorInfo("Constants datetime1 is {0}", Constants.datetime1);
```

[상기 예제의 Log 출력 결과]

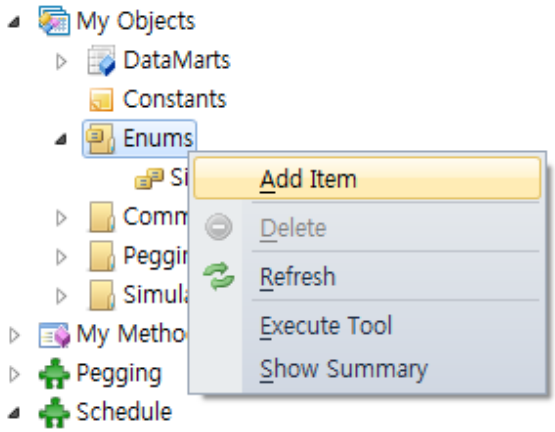


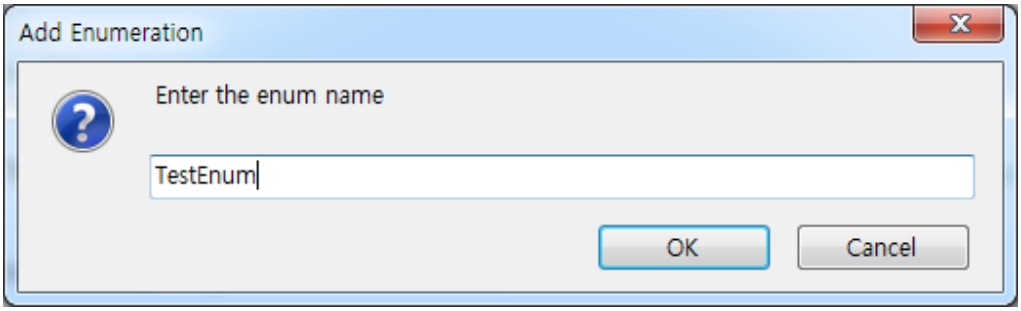
Enum 정의 방법

MOZART IDE 에서 열거형 Data 를 정의하기 위한 기능입니다. MOZART 로 열거형 Data 를 정의할 때에는 My Objects 하위의 Enums 노드에서 팝업메뉴를 통해 정의할 수 있습니다.

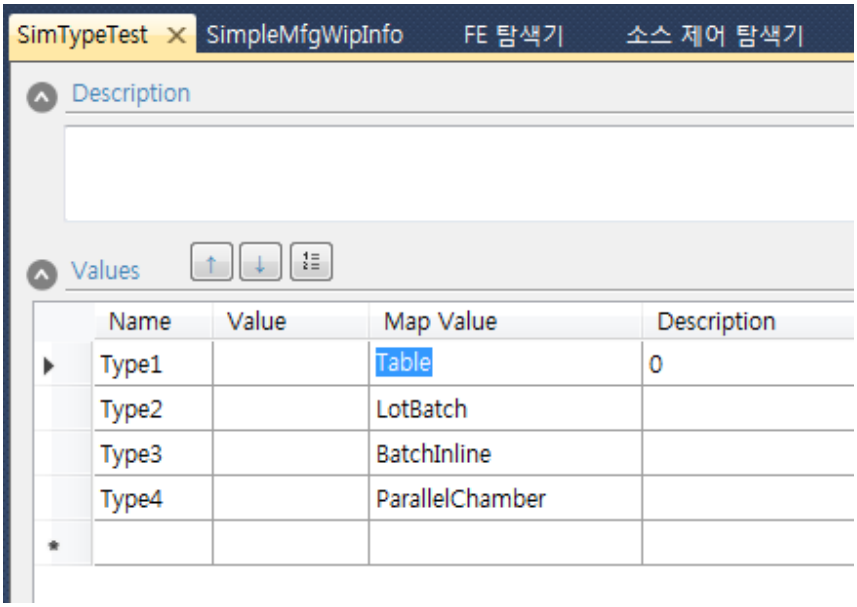
Enum 정의 및 사용방법

1. [**My Objects > Enums**] 노드에서 [**AddItem**] 메뉴를 통해 신규 Enum 타입을 정의합니다.



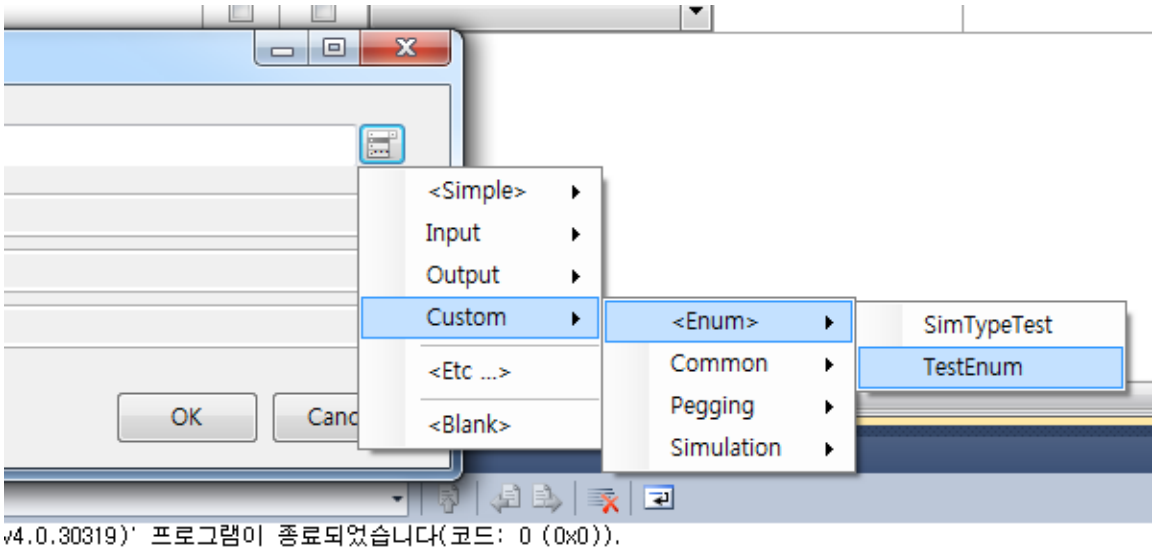


2. Enum 값을 입력한 후 저장합니다. 정의 시에 Value 는 설정을 하지 않아도 상관없습니다.



3. Map Value 의 경우, My Objects 정의 시에 Enum Type 으로 정의된 속성에 Map Field 를 정의할 때 사용할 수 있습니다. Map Value 가 설정된 경우, 설정된 값과 Input 의 Data 가 일치하면 Enum 명으로 자동 변환됩니다. 이때, Input Data 와 일치하는 값은 Enum 명이 아니라 Map Value 여야 한다는 것에 유의합니다. Enum 명과 Input Data 가 일치하더라도 Map Value 가 다르다면 변환되지 않습니다.

4. My Objects 정의 시에 정의한 Enum 을 Property Type 으로 주기 위해서는 아래와 같이 속성의 Property Type 을 **<Other…>**로 설정하여 아래 그림과 같이 Data 타입을 **[Custom > Enum]** 에서 선택합니다.



5. 해당 클래스의 Enum Type 속성이 설정되어 있는 것을 확인할 수 있습니다.

```
var arranges = InputMart.Instance.I
foreach (var arr in arranges)
{
    arr.TestType
    var                                    ent.
```

TestEnum MfgArrange.TestType

Data Mart 사용 방법

Model 의 Inputs / Outputs Schema 및 My Objects 에 정의한 Data Model 클래스에 대한 Collection 을 정의한 DataMart 의 내용을 조회하고 사용자 정의 형식의 Data 를 정의할 수 있는 기능을 제공합니다.

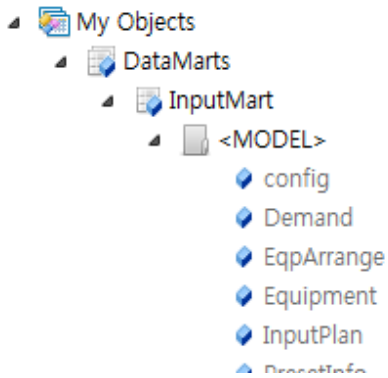
DataMarts 의 구성

- **InputMart** : 시스템 구동 시 사용되는 Inputs 및 중간 Data 저장을 위한 저장소입니다. Model 에서 정의한 Input DataItem 은 Input Persist Config 에서 별도의 처리를 하지 않으면 기본적으로 InputMart 에 EntityTable 형태의 Data 로 생성이 됩니다. My Objects 에서 정의한 Data 클래스에 대한 Collection 도 InputMart 에 생성되며, 사용자가 신규로 저장 형태를 정의하여 사용할 수 있습니다.
- **TempMart** : 시스템 구동 시 사용되는 Data 를 생성하기 위해 Input Loading 시에만 임시로 사용하기 위해 Data 를 저장합니다. Input Persist Config 에서 DataItem 에 대해 “Use temporary context” 옵션을 체크하는 경우, 해당 Data 는 TempMart 에 저장되었다가 Input Loading 이 끝나는 시점에 Memory 에서 삭제됩니다. 사용자가 임시로 사용할 Data 를 추가로 정의할 수 있습니다.
- **OutputMart** : Model 의 Outputs 에 정의된 DataItem 에 대해 EntityWriter 의 형태로 Data 를 저장할 수 있는 Collection 변수가 생성됩니다. 시스템이 구동되는 전체 Life Cycle 동안 접근이 가능합니다. Model 에 지정된 Outputs 만 저장됩니다.

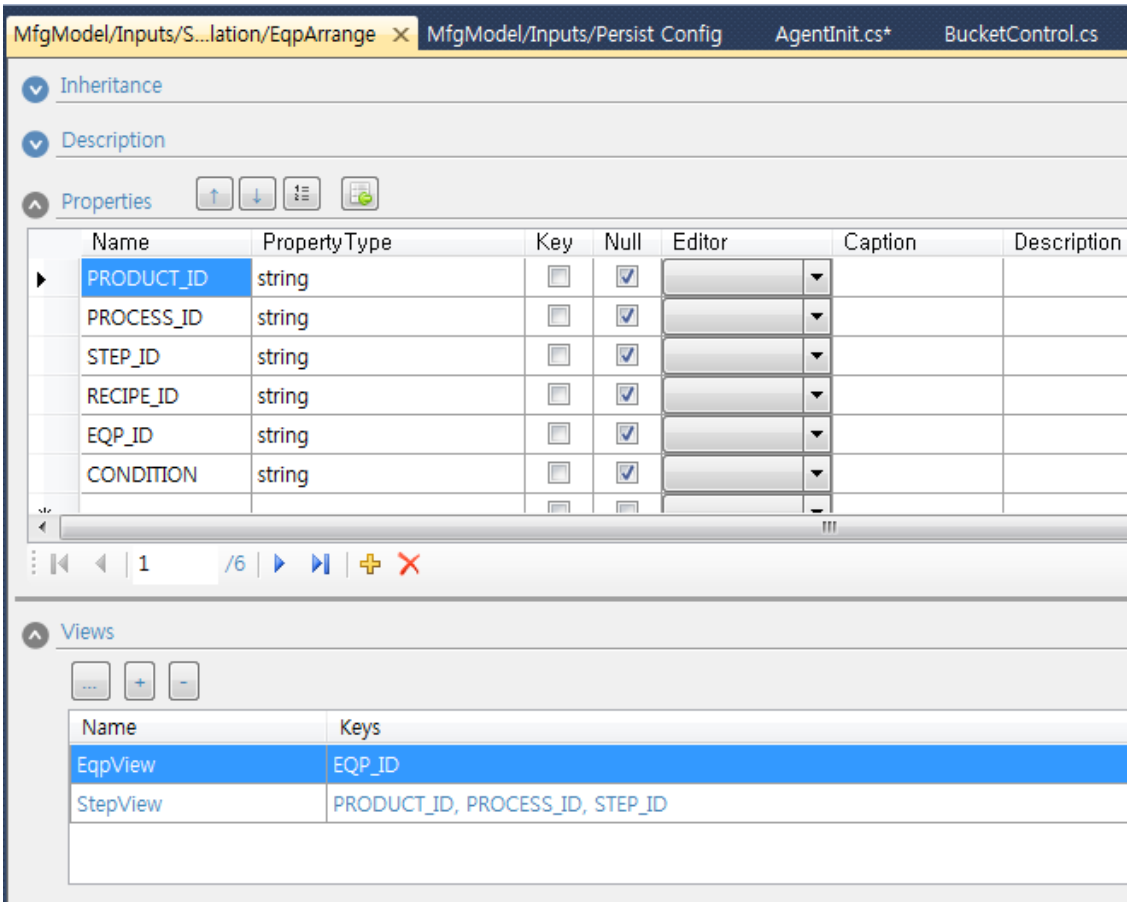
DataMarts 정의 및 사용방법

Model Data 의 저장형태 확인

1. [**DataMarts > InputMart > <Model>**] 노드를 통해 현재 InputMart 에 정의된 Input Data 를 조회할 수 있습니다.



2. 특정 Data 항목을 더블 클릭하면 해당 DataItem 의 Schema 편집화면이 열립니다.



3. InputMart 에는 기본적으로 해당 DataItem Data 클래스(EntityObject) 를 저장하는 EntityTable 이 정의되어 있습니다.

4. 추가적으로 DataItem 에 대한 View 를 지정한 경우, 이 Data 역시 InputMart 에 정의됩니다. 아래는 InputMart 에 정의된 EntityTable 및 View 의 자동 생성 코드를 표시합니다.

```
public EntityTable<EqpArrange> EqpArrange
{
    get
    {
        return this.GetTable<EqpArrange>();
    }
}
private EntityView<EqpArrange> __EqpArrangeEqpView;
public EntityView<EqpArrange> EqpArrangeEqpView
{
    get
    {
        if ((this.__EqpArrangeEqpView == null))
        {
            this.__EqpArrangeEqpView = new EntityView<EqpArrange>(this.EqpArrar
        }
        return this.__EqpArrangeEqpView;
    }
}
```

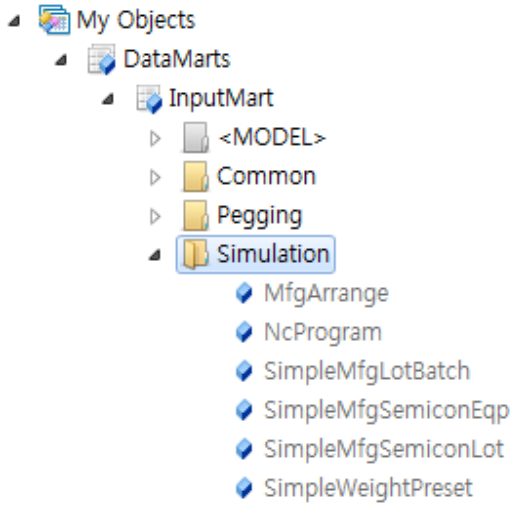
My Objects Data 의 저장형태 확인

1. [**DataMarts > InputMart**] 하위에 My Objects 하위 폴더구조와 동일하게 표시되고 My Object 로 정의한 Data 항목이 표시됩니다.
2. 특정 Data 클래스를 더블 클릭하면 해당 클래스 정의 화면이 열립니다. 화면에서 Collection 을 정의할 수 있습니다. 이에 대한 자세한 사항은 **My Objects 사용법 > My Object 를 InputMart 에 생성하기**를 참조하시기 바랍니다.

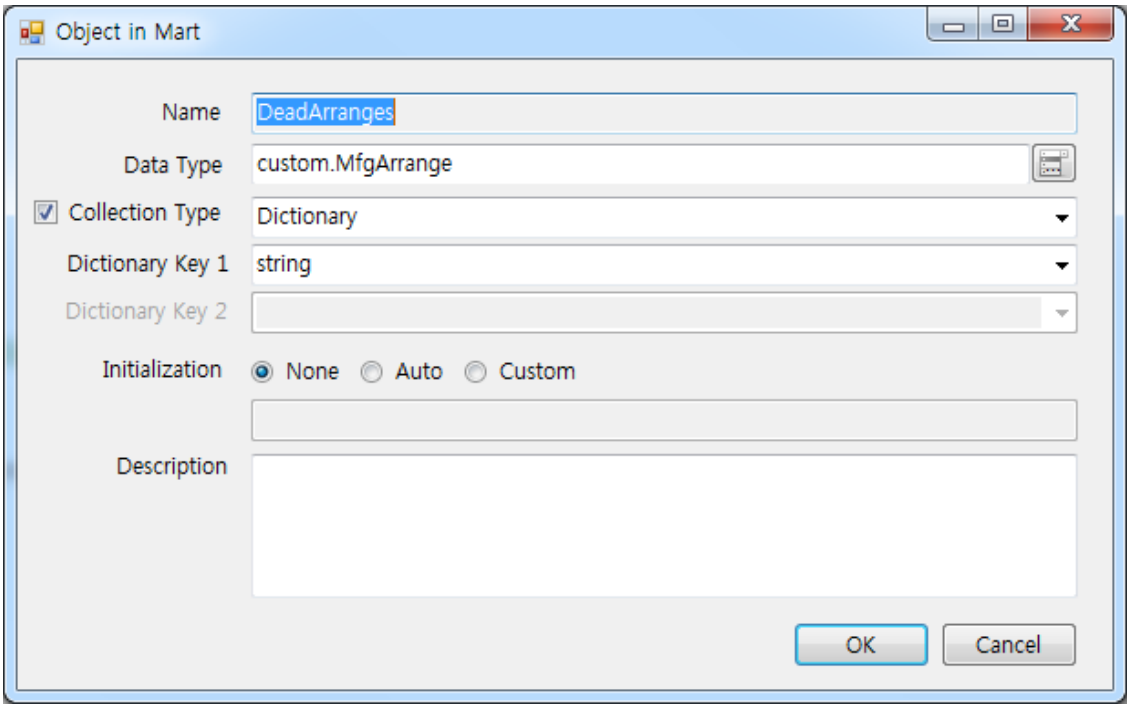
DataMarts 에 사용자 정의 Data Collection 생성

사용자가 Logic 개발 중에 Model 에서 정의한 Data 나 My Objects 로 정의한 Data 클래스 혹은 Library 의 클래스를 사용목적에 따라 추가적으로 저장해두고 사용해야 할 경우가 있습니다. 이 경우에는 DataMarts 의 InputMart, TempMart 에 Data Collection 을 정의하여 사용합니다. 아래는 InputMart 에 신규 Data Collection 을 추가하는 과정이며, TempMart 에도 동일한 방식으로 Data Collection을 정의할 수 있습니다.

1. **[DataMarts > InputMart]** 혹은 하위의 특정 폴더를 선택한 후, **[AddItem]** 메뉴를 실행합니다.



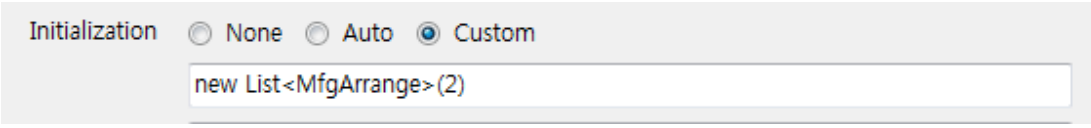
2. DataMarts 에 신규 Collection 을 정의할 수 있는 창에 값을 설정합니다.



- **Data Type** : Collection 에 저장될 Data 타입
- **Collection Type** : 저장 Data 타입
- **Dictionary Key1** : Collection Type 에서 Key 가 필요한 Collection 을 선택한 경우, 첫 번째 Key 의 Data 타입
- **Dictionary Key2** : Collection Type 에서 Key 가 필요한 Collection 을 선택한 경우, 두 번째 Key 의 Data 타입
- **Initialization** : 변수 선언 시에 초기화 코드를 어떻게 작성할지에 대한 옵션

- None : 초기화 코드를 생성하지 않음
- Auto : Collection Type 에 따라 자동으로 변수를 생성함
- Custom : 변수 생성 코드를 하단의 TextBox 에 입력하며, 값 입력 시에는 C# Expression 을 사용함

[입력 예]

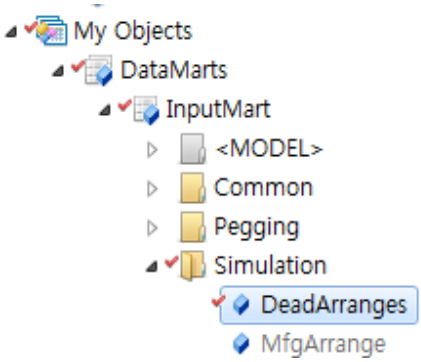


[자동 생성 코드 예]

```
private List<SimpleMfg.DataModel.MfgArrange> __DeadArranges = new List<MfgArrange>(2);
public List<SimpleMfg.DataModel.MfgArrange> DeadArranges
{
    get
    {
```

- **Desciption** : Collection 에 대한 설명

3. 입력 후 저장을 하면, 아래와 같이 DataMart 에 입력한 항목이 생성되며, 전체 저장 혹은 My Objects 를 저장하면 해당 Data 가 InputMart 에 생성됩니다.



4. 정의된 Collection Data 는 InputMart 를 통해 수행 전체 기간 동안 참조가 가능합니다.

Data Collcetion 참조

```
// arrange 의 제약수량이 만족하면 사용 금지 Arrange 에 추가
InputMart.Instance.DeadArrange.Add(arr.Key, arr);
// Arrange 사용 시점에 특정 Key 의 Arrange 가 사용 금지 Arrange 인지 확인
MfgArrange arr;
if (InputMart.Instance.DeadArranges.TryGetValue(arrKey, out arr))
   Logger.MonitorInfo("Arrange key {0} is dead arrange at {1}", arrKey, now);
```

MY METHODS

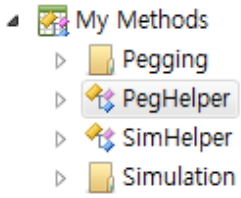
My Method 사용자 가이드

MOZART Main Module 및 Execution Module 에서 공용으로 사용할 수 있는 Logic 처리 함수를 정의하기 위한 기능입니다. 사용자 조직의 Business Logic 을 구현할 때 자주 재사용되는 함수를 등록하여 사용하는 개념으로 이해할 수 있습니다. My Method 는 사용방식이 C# 언어로 Visual Studio 의 Solution Explorer 를 통해 작업하는 것과 유사합니다.

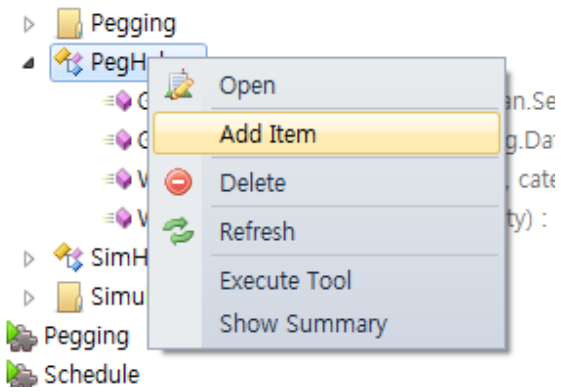
My Method 정의 및 사용 방법

Method Group 및 Method 추가하기

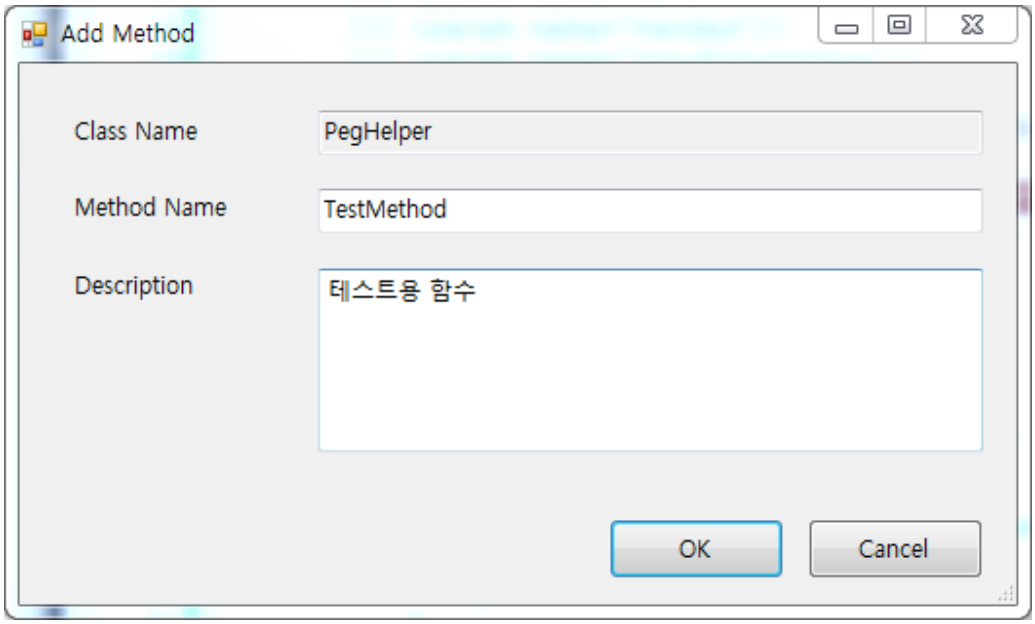
1. My Method 노드에서 팝업메뉴 통해 Mehtod 집합을 구성하는 Folder (**Add Folder** 메뉴)나 Method 집합이 되는 Method Item (**Add Item** 메뉴)을 추가합니다.



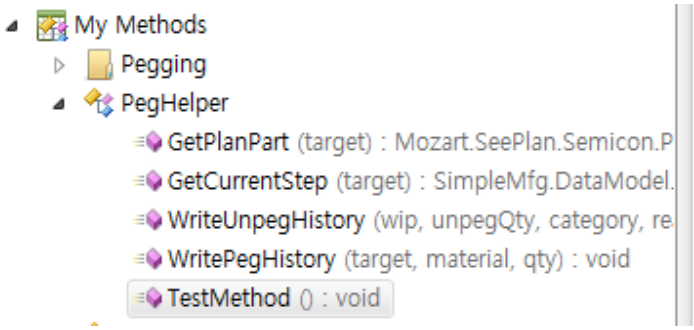
2. Method Item 은 클래스와 Mapping 되며 실제 자동 생성되면서 개별 클래스가 생성됩니다.
3. Method Item 에 Method 를 추가합니다.



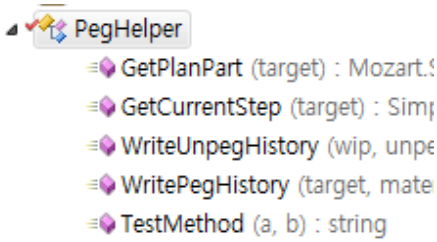
4. Method 추가 시에는 Method 명과 Description 을 입력합니다.



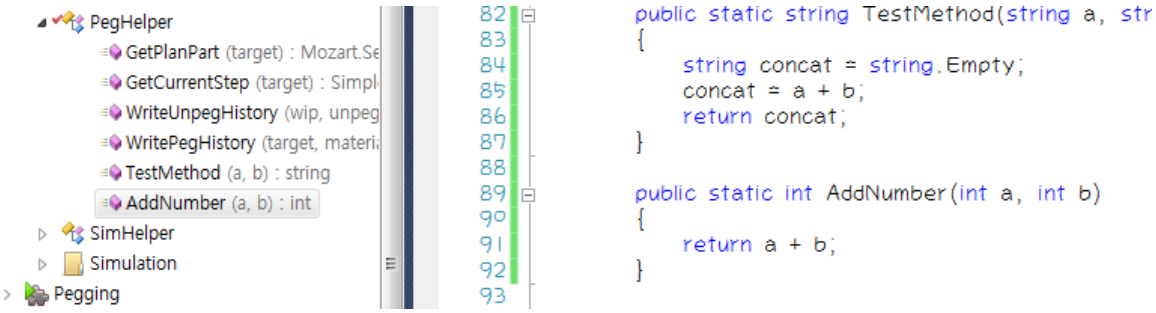
5. **[OK]** 버튼을 누르면 My Method 의 선택한 클래스에 아래와 같이 Method 가 추가됩니다.



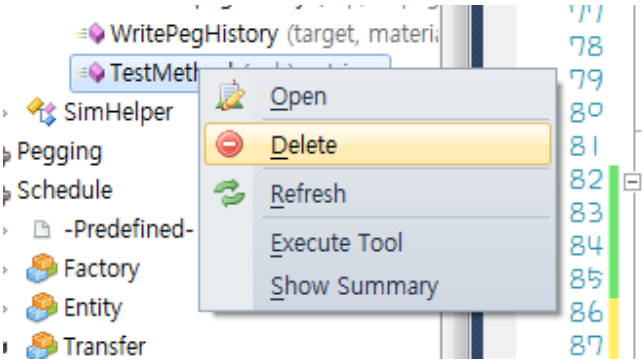
6. 해당 Method 를 더블 클릭하면 함수 원형이 아래와 같이 생성됩니다.
7. 여기서 Return 값 및 Arguments 등을 입력하고 함수 본문을 작성하면 해당 내용이 Tree 노드와 동기화됩니다.



8. 역으로 소스에 함수를 추가하는 경우에도 My Methods 의 Tree 노드에 Method 가 동기화되어 자동 생성됩니다.



9. 작성된 Method 는 해당 Method 노드에서 [Delete] 팝업메뉴로 삭제할 수 있습니다. 삭제 시 소스에서도 Method가 삭제됩니다.



MAIN MODULE

Main 모듈 개요 및 실행 구조

Main 모듈 개요 및 실행 구조

MOZART 의 실행 프레임워크에서는 IModelExecutor Interface 를 구현한 실행 Module(Execution Module) 이나 Main Module 만 실행 가능합니다. SEEPLAN Library 에 의해 기본 제공되는 독립적인 실행 Module 은 Pegging, Loading Simulation 두 가지입니다. 각 Module 은 Core Model 의 구동을 통해 기본적인 작동을 하며 Feature Extension 구조를 통해 외부에서 제어할 수 있는 확장 구조를 가집니다. Library 기반으로 하는 Project 에서는 확장구조를 이용해 Model 을 Customizing 할 수 있습니다. 이에 대한 자세한 사항은 **Execution Module Library 개요**를 참고하시기 바랍니다.

Main Module 은 MOZART 를 통해 실행하고자 하는 Model 에 대한 속성을 설정하거나, Model 실행 옵션을 조정하고 Model 실행 전/후에 처리할 사용자 처리 Logic 을 정의할 수 있도록 구성되어 있습니다. 즉, Main 은 프로그램의 주 진입점에 대한 Customizing 을 가능하게 해주는 기본 실행 Module 입니다. 또한 Domain Library 를 기반으로 생성된 경우, Library 에서 제공하는 기본 Configuration 정보를 설정할 수 있습니다.

Configuration

Library 에서 제공하는 기본적인 Parameters 를 설정할 수 있습니다. 개별 Library 의 설정과 관련된 부분은 Execution Module 을 참고하시기 바랍니다.

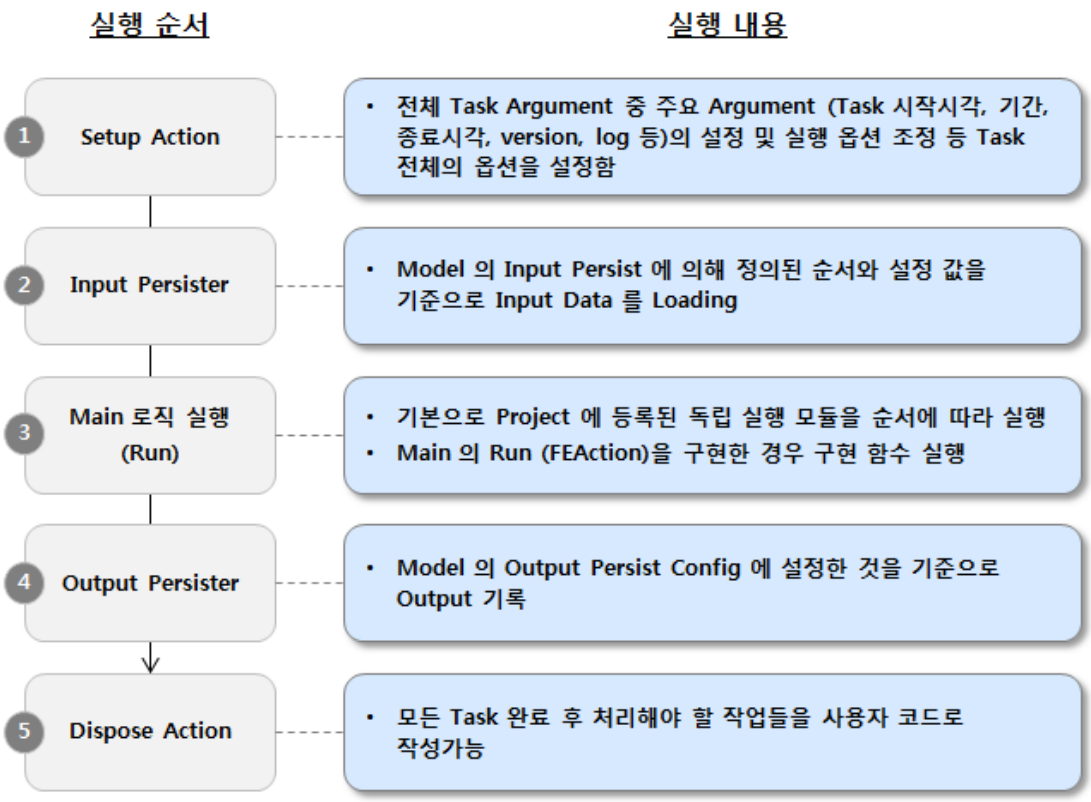
1. MOZART Explorer 의 [**SeePlan Config**] 노드를 더블 클릭하면, 참조된 Library 에서 제공하는 Parameters 설정 화면이 활성화됩니다.
2. Configuration 창의 왼쪽 Tree View 에서 설정이 가능한 모든 Parameter Page 를 조회할 수 있습니다.
3. 특정 Tree 노드를 선택하면, 해당 노드에 해당하는 Parameter Page 가 화면에 로드되고 Parameters 의 값을 설정 할 수 있습니다.

Main Logic Control

Main Logic 의 경우 MOZART Task 구동 Logic 을 사용자화 할 수 있도록 정의된 FEAction 을 노출시킵니다. Main Module 의 FEModel 에 대한 설명은 **Main Control** 을 참고하시기 바라며, 해당 Logic 의 개발 방식은 **FE Control Logic 개발방법** 을 참고하시기 바랍니다.

Main 모듈 실행 구조

Main Module 은 MOZART 로 구현된 모든 Task 를 구동 시키는 Main 실행 Module 에 대한 FEComponent 를 노출합니다. MOZART Main 실행 Module 은 아래 그림과 같은 순서로 실행됩니다.



Server 에서 일정 주기로 수행할 작업 중 Simulation, Pegging 과 같이 Library 를 사용하지 않는 단순한 Logic 은 Main Module 과 Model 의 기능만으로 쉽고 빠르게 구현이 가능합니다. FEModel 에 대한 상세 정보는 [FEModel/Main Control]을 참고하시기 바랍니다.

Configuration

Configuration 은 Library 에서 정의된 시스템 Parameters 를 설정하기 위한 기능입니다. Project 생성 시 선택한 Library 에 따라 Configuration 항목이 변경됩니다.

Configuration 설정 방법

1. MOZART Explorer 의 [**SeePlan Config**] 노드를 더블 클릭하거나 팝업메뉴 중 [**Open**]을 클릭하면, 참조된 Library 에서 제공하는 Parameters 설정 화면이 활성화됩니다.

2. Configuration 창의 왼쪽 Tree View 에서 설정이 가능한 모든 Parameter Page 를 조회 할 수 있습니다.
3. 특정 Tree 노드를 선택하면, 해당 노드에 해당하는 Parameter Page 가 화면에 로드되고 Parameters 의 값을 설정 할 수 있습니다.

Configration 설정 값 참조 방법

Configruation 설정 값은 기본적으로 Library 에서 참조하지만 해당 입력값을 사용자 Logic 에서 사용할 필요가 있는 경우 소스 코드 상에서 Configuration Tree 의 Library 명칭 +"Configuration" 의 형태로 Service 를 요청하여 참조할 수 있습니다. 기본 참조 Library 가 SEEPLAN, SEMICON Library 인 경우 SeePlanConfiguration, SemiconConfiguration 과 같은 형태로 Configuration Service 를 요청하여 참조할 수 있습니다. 아래는 참조 방식에 대한 소스 코드 예입니다.

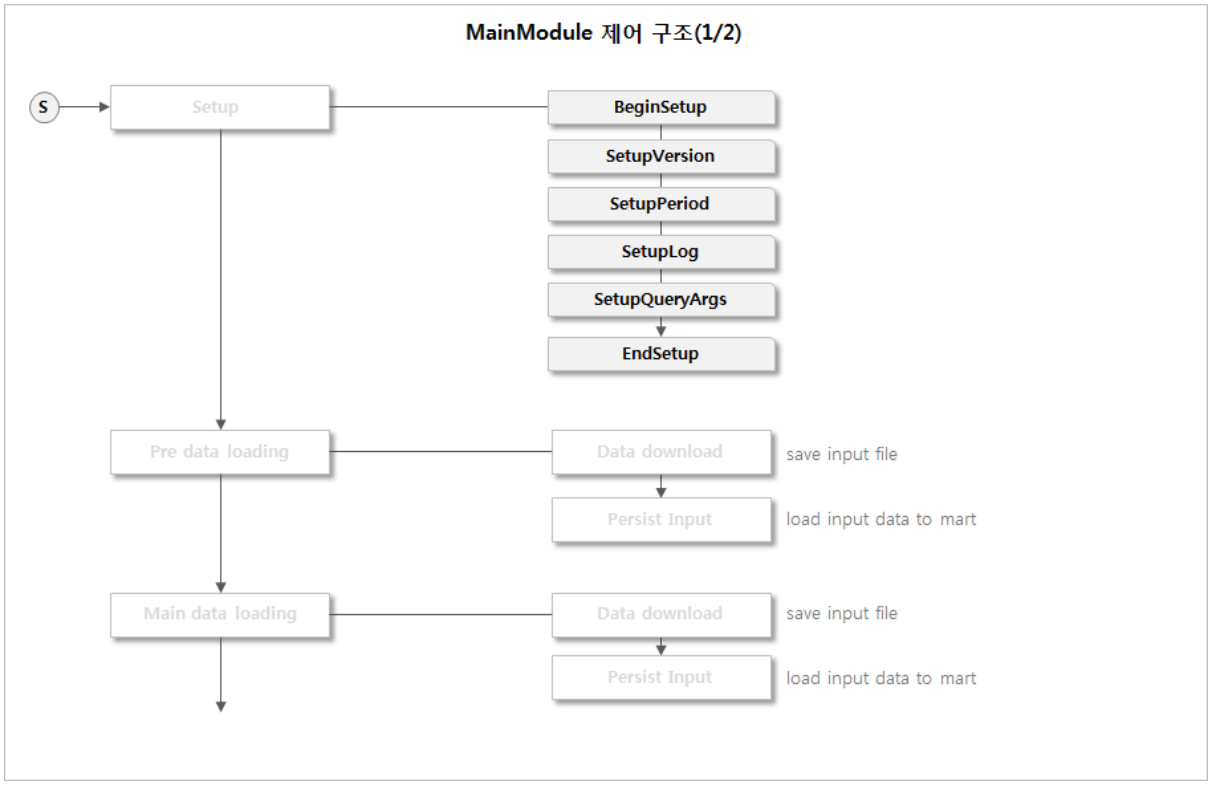
Main Module 의 Configuration 값 참조 방법

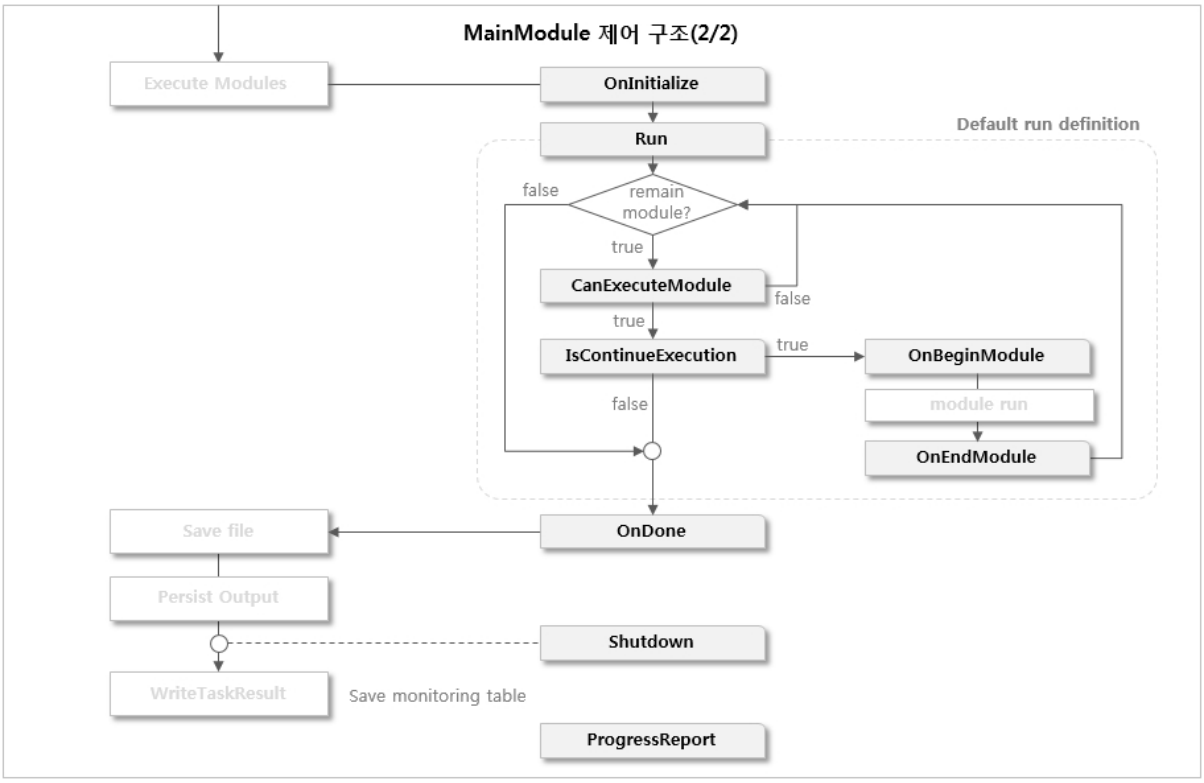
```
// SEEPLAN Library Configuration 사용예
// SEEPLAN Library 의 LotUnitSize, DefaultSetupTime Config 항목 참조방법
int defaultLotUnitSize = SeeplanConfiguration.Instance.LotUnitSize;
float defaultSetupTime = SeeplanConfiguration.Instance.SetupTimeMinutes;

// SEMICON Library Configuration 사용예
// SEMICON Library 의 경우 ApplyNoArrangeMove 와 같은 Boolean Type Config 항목 참조방법
bool applyNoArrangeMoveOption = SemiconConfiguration.Instance.NoArrangeMove;
if (applyNoArrangeMoveOption)
{
    // to do
}
```

Main 컨트롤

MOZART 의 Model 은 Library 에서 제공하는 Model Task 에 의해 구동되며, Main Control 은 Model Task 가 Model 을 구동시키기 위한 Parameters 를 설정하거나, Model 구동이 끝난 후 사용자에 의해 추가적인 Data 처리를 수행할 수 있도록 지원하는 FEComponent 입니다.

Model Task 는 Setup 단계를 통해 Task 구동 시작/완료시각 및 Simulation Version 등을 설정하고 그 외 시뮬레이션 수행 시 Model 에서 사용할 정보를 설정합니다. Model 의 실행은 기본적으로 MOZART IDE 를 통해 등록된 실행 Module 을 순서에 따라 실행하도록 구현되어 있습니다. Forward, Backward Planning 실행을 담당하는 Pegging & Loading Simulation Module 이 기본 제공되는 실행 Module 입니다. Module 의 실행순서는 기본적으로 Pegging 후 Simulation 을 수행하도록 설정되어 있습니다.





Setup Phase

전체 작업수행 방식을 조정할 수 있도록 구성된 FEAction 의 집합입니다. 실행 Model 의 지정 및 순서조정을 포함한 모든 설정이 가능합니다.

1. **BeginSetup**: Setup 처리 직전 Model 수행 시 필요한 정보를 생성하거나 업데이트가 필요한 경우 이를 처리하는 Logic 을 구현합니다.
2. **SetupVersion**: Setup 수행 시점에 호출되며, Plan Version 을 사용자 정의 방식으로 설정할 수 있습니다.
3. **SetupPeriod**: Setup 수행 시점에 호출되며, Plan 기간을 사용자 정의 방식으로 설정할 수 있습니다.
4. **SetupLog**: Setup 에 대한 Log 를 사용자 정의 방식으로 구현할 수 있습니다.
5. **SetupQueryArgs** : In/Output 을 위한 DataAction 실행 시 사용할 수 있는 Arguments 를 설정할 수 있습니다.
6. **EndSetup**: Setup 이 끝난 직후 호출되며, 정상적인 Setup 처리 Logic 외에 추가적인 설정이 필요한 경우 본 FEAction 을 통해 구현합니다.

Input Data Download & Loading Phase

Module 의 실행을 위해 Data Source 로부터 Data 를 Download(Query) 하여 파일로 저장하고 Memory 의 InputMart, TempMart 에 Data 를 로딩하는 단계입니다. 별도의 사용자 정의 Logic 을 수행할 수 있는 Event 는 없습니다.

- **Preloading** : Input Data 중 Preloading 설정되어 있는 Data 에 대해 Input File Download 및 Data Loading 을 수행합니다. Main Input Data Loading 을 수행하거나 전체 실행을 제어하기 위한 조건정보를 Data Source 로부터 로딩하는 용도로 주로 사용됩니다.
- **Main Input loading** : Input Data 중 Preloading 설정되어 있지 않은 Data 에 대해 Input File Download 및 Data Loading 을 수행합니다.

Execute Phase

MOZART 에서 실행을 제어하며 각 실행 Module 의 초기화 및 종료시점 Event 에 대한 처리를 사용자화 할 수 있습니다.

1. **OnInitialize** : Input Data Loading 이후 Task 를 수행하기 직전에 호출됩니다. 로딩된 Data 를 1차로 가공하여 추가적인 Input Data 를 만드는 용도로 사용할 수 있습니다.
2. **Run** : Task 수행의 진입점이 되는 Main 함수입니다. 본 함수는 Model 에 포함된 모든 실행 Module 을 Pegging, Simulation Module 의 순서로 실행시키도록 기본 구현되어 있는 Definition 이 Binding 됩니다. Pegging, Simulation Module 을 추가하는 경우에는 기본 Definition을 사용할 수 있습니다. 만일 등록된 Module 을 임의의 방식으로 실행하고자 하는 경우에는 Run 함수를 재정의할 수 있습니다.

Run 함수의 구현을 통한 Module 실행 제어

```
// CustomTest Module 을 추가한 후 Run 함수를 재정의한 예제
// 일반적으로 Custom Module 이 Simulation 이전 우선순위이나
// Simulation 결과를 재가공하는 Module 을 추가해야 하는 경우 아래와 같이 구현할 수 있습니다.
public void RUN1(ModelContext context, ref bool handled)
{
    // Module 명으로 Simulation Module 을 참조(Module 명은 Tree 에 표시된 명칭과 같음)
    var module = context.GetExecutionModule("Schedule");
    if (module !=null)
    {
        module.Execute(context) ;
        if (context.HasErrors)
            return;
    }
    // Custom Module 참조
    module = context.GetExecutionModule("CustomTest");
    if (module != null)
```

```
    {
        module.Execute(context);
        if (context.HasErrors)
            return;
    }
}
// Custom Module 의 실행 함수 구현예제
public void EXECUTE0(ModelContext context, ref bool handled)
{
    // Simple 한 Log 출력예제
    Logger.MonitorInfo("This is Custom Module Test!");
    // InputMart, OutputMart 의 Data 를 사용하여 Logic 을 구현할 수 있습니다.
}
```

실행 결과 아래와 같이 시뮬레이션 수행 이후에 Custom Module 의 수행 Log 를 남기도록 합니다.

```
Elapsed Time = 00:00:03.9050451
  Simulation-Init...2015-07-22 20:35:32
  > Elapsed Time = 00:00:00.8080581
  Simulation-Run ...2015-07-22 20:35:33
shift changed at 20150601 060000 ..... 00:00:00.0340996
shift changed at 20150601 140000 ..... 00:00:00.7151278
shift changed at 20150601 220000 ..... 00:00:00.1676605
shift changed at 20150602 060000 ..... 00:00:00.1513568
  > Elapsed Time = 00:00:01.2099895
This is Custom Module Test!
Disposing ...2015-07-22 20:35:40
Elapsed Time = 00:00:00.0109289
Done ...2015-07-22 20:35:40
```

RUN_DEF 구현 코드

```
public virtual void RUN_DEF(ModelContext context, ref bool handled)
{
    var handler = TaskControl.Instance;
    var modules = context.GetOrderedExecutionModules().ToArray();
    Logger.StartHandler(context.GetLog(MConstants.LoggerExecution));
    try
    {
        int count = modules.Length;
        for (int i = 0; i < count; i++)
        {
            var module = modules[i];
            if (!handler.CanExecute(module, context))
                continue;
            if (!handler.IsContinueExecution(module, context))
                break;
            Logger.MonitorInfo(module.Name + " Start.");
            this.lastResult = module.Execute(context);
            if (context.HasErrors)
                break;
            Logger.MonitorInfo(module.Name + " End.");
        }
    }
    finally
```

```
    {
        Logger.EndHandler();
    }
}
```

1. **CanExecuteModule** : Run 함수의 기본 Definition 에서 사용되는 함수입니다. MOZART Project 에 등록된 Module 별로 함수가 호출되며, 해당 Module 의 실행 여부를 판별하도록 Logic 을 구현합니다. 예를 들어 Pegging, Simulation Module 이 모두 추가된 상태에서 Pegging 만 진행하고자 할 경우 본 함수에서 Simulation Module 의 결과값을 false 로 반환하여 Pegging 만 수행하도록 처리할 수 있습니다.
2. **IsContinueExecution** : Run 함수의 기본 Definition 에서 사용되는 함수입니다. 본 함수는 Module 실행의 중단 여부를 판별합니다. 반환값이 false 인 경우 Module 의 실행을 종료합니다. 실행되지 않은 잔여 Module 이 몇 개이건 상관없이 실행을 중지합니다.
3. **OnBeginModule** : Run 함수의 기본 Definition 에서 사용되는 함수입니다. 실행 Module 의 초기화 시점에 호출되어 사용자 정의 Logic 을 구현할 수 있습니다. 모든 실행 Modeule 의 실행 시작시점에 호출됨으로 Model 에 따라 Logic 을 구분해서 구현해야 합니다.
4. **OnEndModule** : Run 함수의 기본 Definition 에서 사용되는 함수입니다. 실행 Modeule 의 실행이 끝난 시점에 호출되어 사용자 정의 Logic 을 구현할 수 있습니다. 모든 실행 Module 의 실행 종료시점에 호출되므로 Model 에 따라 Logic 을 구분해서 구현해야 합니다. 만일 실행 Module 의 수행 시점에 오류가 발생하는 경우에도 OnEndModule 함수는 호출됩니다. 여기서 발생된 오류에 대한 로딩작업을 별도로 수행할 수 있습니다.
5. **OnDone** : Project 에 등록된 모든 Module 의 실행이 종료되는 시점에 호출됩니다. 결과 기록 직전 시점으로 결과에 대한 추가적인 가공이 가능합니다.

End Phase

Save output file & Save DB : Memory 상(OutputMart) 결과 Data 를 Output File 에 기록한 이후 결과 중 DB 에 업데이트가 필요한 항목에 대해 처리합니다.

1. **Shutdown**: Model 실행이 완전히 종료된 직후 후속 처리가 필요한 경우 이를 구현할 수 있습니다. 기본적으로 Model 작업 수행 중 발생하는 오류 Log 를 남기고자 하는 경우에 사용이 가능합니다. Output Persist Config 의 Monitoring table 을 설정한 경우 아래의 사용예시와 같이 코딩할 수 있습니다.

Task 실행 로그 기록 예제

사전에 [**Persist Config > Output Config**] 의 Monitoring table 에 최종 결과를 저장할 Outputs 을 설정한 후 아래와 같이 Shutdown 함수를 구현합니다. 최종 결과 저장 시 문제를 고려하지 않는다면, 해당 Data 를 OnDone 혹은 OnEndModule 에서 기록할 수도 있습니다. 단, Monitoring table 로 설정된 경우 Shutdown 이후에 File, DB 에 저장됩니다.

```
public void SHUTDOWN_0(ModelTask task, ref bool handled)
{
    // 결과 Output DataItem 명이 ResultFlag 인 경우
    ResultFlag result = new ResultFlag() ;
    result.STATE_TIME = DateTime.Now ;
    result.STATE = task.HasError ? "FAIL" : "SUCCESS" ;

    // 해당 결과 Table 에 값 추가
    OutputMart.Instance.ResultFlag.Add(result) ;
}
```

최종 실행 결과 모니터링 결과 저장 : Task 수행 최종 결과를 기록합니다. Output Persist Config 에 설정된 Monitoring table 의 값을 저장합니다.

1. **ProgressReport**: DB 읽기, Data 로딩, Module 실행, Data 기록, DB 쓰기 등을 시작하고 끝낼 때마다 호출되는 Action 입니다. Action 이 실행되는 시점을 구분하는 값은 ‘Stage’ 이며, ‘Stage’ 의 이름에 해당하는 시점에서 ProgressReport Action 이 호출됩니다.

   ProgressReport 의 ‘**Stage**’ 입력 값

   - **PreDownload_Start** : Preloading 대상 DB 읽기 시작
   - **PreDownload_End** : Preloading 대상 DB 읽기 끝
   - **PreLoading_Start** : Preloading 대상 Data 로딩 시작
   - **PreLoading_End** : Preloading 대상 Data 로딩 시작
   - **AutoDownload_Start** : DB 읽기 시작
   - **AutoDownload_End** : DB 읽기 끝
   - **DataLoading_Start** : Data 로딩 시작
   - **DataLoading_End** : Data 로딩 끝
   - **{모듈(Pegging/Simulation/Custom) 이름} Module_Start** : Module 실행 시작
   - **{모듈(Pegging/Simulation/Custom) 이름} Module_End** : Module 실행 끝
   - **SaveOutput_Start** : Data 기록 시작
   - **SaveOutput_End** : Data 기록 끝
   - **CommitOutput_Start** : DB 쓰기 시작

- **CommitOutput_End** : DB 쓰기 끝
- **SaveMonitoringOutput_Start** : Monitoring Table Data 기록 시작
- **SaveMonitoringOutput_End** : Monitoring Table Data 기록 끝
- **CommitMonitoringOutput_Start** : Monitoring Table DB 쓰기 시작
- **CommitMonitoringOutput_End** : Monitoring Table DB 쓰기 끝

2. **ModelContext 클래스** : Main Control 의 사용자 정의 Logic 구현 시 사용할 수 있는 기본 클래스인 ModelContext 의 참조 항목에 대한 설명입니다.

   주요 속성 & Method

   - **StartTime** : Model 수행 시작시각 (Arguments 의 start-time 설정 시각)
   - **EndTime** : Model 수행 종료시각 (Arguments 의 start-time + period)
   - **Arguments** : Input Arguments 설정 값 Collection
   - **QueryArgs** : Input, Output persist 시 사용되는 DataAction 에서 사용될 Arguments 설정값 Collection
   - **Version** : 실행 Version 의 VersionInfo 객체
   - **VersionNo** : 실행 Version 번호 문자열 (이름, 실행시각, VersionNo 등의 속성을 가집니다.)
   - **VModelName** : 실행 Vmodel 의 명칭
   - **HasError** : Model 실행 Error 발생 여부
   - **LastException** : Model 실행 시 발생한 에러 (Exception) 에 대한 정보
   - **Result** : Object 형태의 Task 수행 결과입니다. Dependent Trigger 를 생성할 때 참조되는 Task 의 결과 기록
   - **GetExecutionModule(string moduleName)** : 지정한 Module 명으로 실행 Module 반
   - **GetOrderedExecutionModules()** : 등록된 전체 Module List 반환

참조 방법

- **Main Module 의 Action Parameter 로 참조** : Main Module 의 Action 구현 시점에 사용하며, ModelTask.Context 를 통해 접근 가능합니다.
- **ModelContext.Current** 으로 참조 : 임의 시점에 사용할 수 있습니다.

FE Control 로직 개발 방법

Core Model 을 제어하기 위한 Logic 개발 지점은 FEComponent 를 통해 제공됩니다. Simulation 과 Pegging Logic 개발 시 MOZART Explorer 에 각 Model 의 1Level 노드가 생성되며 2Level 로 Feature Extension Category, 3Level 로 FEComponent 를 조회할 수 있습니다. FEComponent 하위에 해당 Component 가 제공하는 FEAction 노드가 표시됩니다. 사용자는 FEAction 의 역할에 맞는 사용자 정의 Logic을 추가할 수 있습니다. 이러한 사용자가 개발한 FEAction 의 실행 코드를 FEDefinition 이라고 하며, FEAction의 하위 노드로 생성됩니다.

FEDefinition 생성하기

FEDefinition 은 아래의 절차로 생성할 수 있습니다.

1. 개발하려는 Model 의 FEAction 을 찾습니다. 이때, Logic 개발 지점을 찾기 위해 FEDocument 를 이용할 수 있습니다.
2. FEAction 노드를 선택하고 팝업메뉴에서 [**Add**] 버튼을 클릭합니다.
3. Add FE Method Dialog 에 각 항목을 입력하여 FEDefinition 을 정의합니다.
   - **Category Name** : FEDefinition 의 기능 구분입니다. 사용자가 입력하여 정의할 수 있으며 Method 의 코드가 생성되는 클래스 명이 됩니다. 기존에 등록된 클래스 명을 선택할 수 있으며 새로 입력하여 정의할 수 있습니다.

     > 💡 **Note**
     > Category Name 을 잘 분류하여 사용하면, Project 에서 기능 단위의 진도 및 이력 관리를 용이하게 할 수 있습니다. 예를 들어 사용자가 요구한 특정 기능에 대한 개발을 할 경우에 Category 를 해당 요구 기능 명칭으로 정의한 후 관련 Definition 을 정의하는 경우 기능에 대한 구현 사항을 하나의 클래스에서 관리할 수 있으며, TFS 와 연결해서 관리하는 경우 기능에 대한 이력 관리가 가능합니다.

   - **Method Name** : 자동 생성되는 함수의 이름입니다.

- **Description** : 함수에 대한 설명입니다.
- **Enable** : FEDefinition 의 활성화 여부입니다. 활성화 설정된 FEDefinition 만 최종 운영 Model 생성 시 반영됩니다.
- **Activation Condition** : 해당 FEDefinition 의 활성화 조건입니다. Enable 상태라도 Activation Condition 이 설정된 경우 이를 만족해야 운영 Model 생성 시 코드가 반영됩니다. Activation Condition 은 Arguments 에서 정의된 Boolean Type의 Parameters 중에 선택할 수 있습니다.

4. Method 정보를 모두 입력하면 Solution Explorer 의 Logic 폴더에 Method 정의 시 설정한 Category Name 으로 클래스가 생성되며, 소스 편집기에 해당 .cs 파일이 열리고 정의한 Method 를 편집할 수 있도록 Focus 가 이동합니다.
5. 해당 Method 를 C# 문법에 따라 구현합니다.

> 💡 **Note**
> FEAction 에 따라 자동으로 생성되는 FEDefinition 의 함수 원형에 다음과 같은 사전 정의된 두 가지의 Arguments 가 포함됩니다.

- **handled** : FEAction 에 Binding 된 FEDefinition 의 실행을 제어하는 변수로 해당 변수를 true로 설정하게 되면 현재 FEDefinition 종료 후 FEAction 을 종료합니다. 그렇지 않은 경우 FEAction 에 Binding 된 모든 FEDefinition 을 실행하도록 운영 Model 이 생성됩니다.
- **prevReturnValue** : 이전 실행된 FEDefinition 의 결과값입니다. FEAction 내의 FEDefinition 들이 다른 Logic 으로 같은 Operation 을 수행하는 경우 선행 결과가 후행 Logic 의 Input 으로 반영이 필요한 경우 이를 코드에서 반영하여 사용합니다.

FEDefinition method code sample

```
public IList<string> GET_LOADABLE_EQP_LIST(DispatchingAgent da, ISimEntity entity,
                            ref bool handled, IList<string> prevReturnValue)
{
   InputMart im = ServiceLocator.Resolve<InputMart>();
   EntityView<EqpArrange> view = im.EqpArrange.AsEntityView();
   object[] obj = { "LINE_ID", "PRODUCT_ID", "STEP_ID" };
   IEnumerable<EqpArrange> arrs = view.FindRows(obj);
   List<string> list = new List<string>();
   foreach (EqpArrange arr in arrs)
   {
       if (!list.Contains(arr.EQPID))
           list.Add(arr.EQPID);
   }
   return list;
}
```

FEAction 편집하기

FEAction 과 FEDefinition 은 1:N 의 관계를 가집니다. 즉, 하나의 Logic Point 에 복수의 Logic 을 구현할 수 있습니다. 예를 들어 Dispatching 시점에 재공을 필터링하는 Point 에서는 필터링 제약조건의 수만큼 FEDefinition 이 추가될 수 있습니다. 따라서 FEAction 에 Binding 된 FEDefinition 이 여러 개인 경우 FEDefinition 간 순서를 지정할 수 있어야 하며, 시간이 지나면서 Logic 이 변경되고 추가/삭제 되면 Binding 된 FEDefinition 역시 삭제되거나 변경될 수 있습니다. FEAction 의 [**Edit**] 메뉴를 통해 FEAction 과 연계된 FEDefition 에 대해 일괄 편집할 수 있습니다.

1. 편집할 FEAction 노드를 클릭하고 팝업메뉴에서 [**Edit**] 버튼을 클릭합니다.
2. Bind Dialog 가 로드됩니다.
3. Binding List 의 순서를 조정하기 위해서는 우측 Binding List 리스트박스의 상단의 ⬆️ ⬇️ 버튼을 사용합니다.
   - Binding List 에서 순서를 변경할 List 항목을 선택한 후
   - 순서 조정 버튼으로 순서를 위/아래로 조정합니다.
4. Bindable List 에는 선택된 FEAction 에 Binding 이 가능한 FEDefinition 의 List 를 조회할 수 있습니다. 이 List 중 추가로 Binding 을 하고자 할 때는 Bindable List 의 항목을 더블 클릭하면 List 항목이 Binding List 에 추가됩니다.
5. Binding List 에서 List 항목을 더블 클릭하면 Binding 이 해제됩니다. 아이템은 Binding List 에서 삭제되고 Bindable List 에 추가됩니다.
6. Binding List 의 각 행 별로 Enable Check 버튼을 통해 FEDefinition 의 활성상태를 변경할 수 있습니다.
7. 편집이 모두 완료된 후에 [**OK**] 버튼을 눌러 변경내용을 저장합니다. [**OK**] 버튼을 누르지 않으면 변경내용이 반영되지 않습니다.

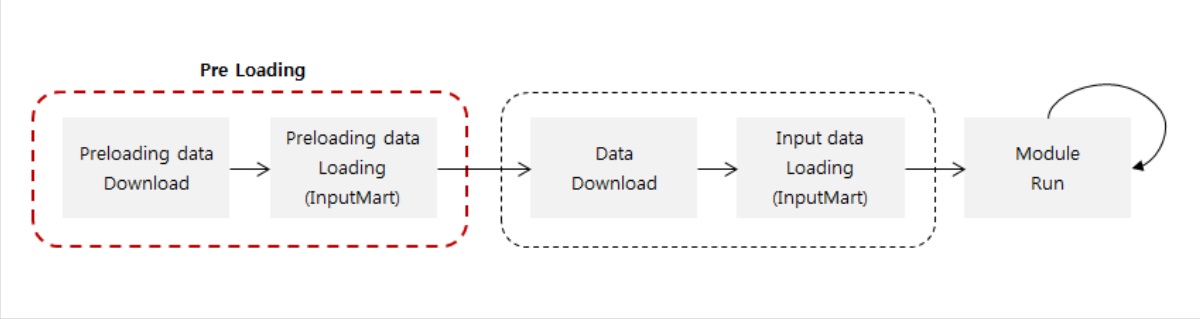
FEDocument 참조하기

FEAction 및 FEComponent 에 대한 설명이 필요한 경우 각 노드의 팝업메뉴에서 [**Show Summary**] 버튼을 사용해서 해당 아이템의 설명을 조회할 수 있습니다. 이때 Document Explorer 가 활성화됩니다. Document Explorer 를 닫지 않고 MOZART Explorer 의

FEComponent 와 FEAction 을 Navigation 하면 선택된 항목에 대한 설명이 Document Explorer 와 동기화되어 변경됩니다.

Data Preloading

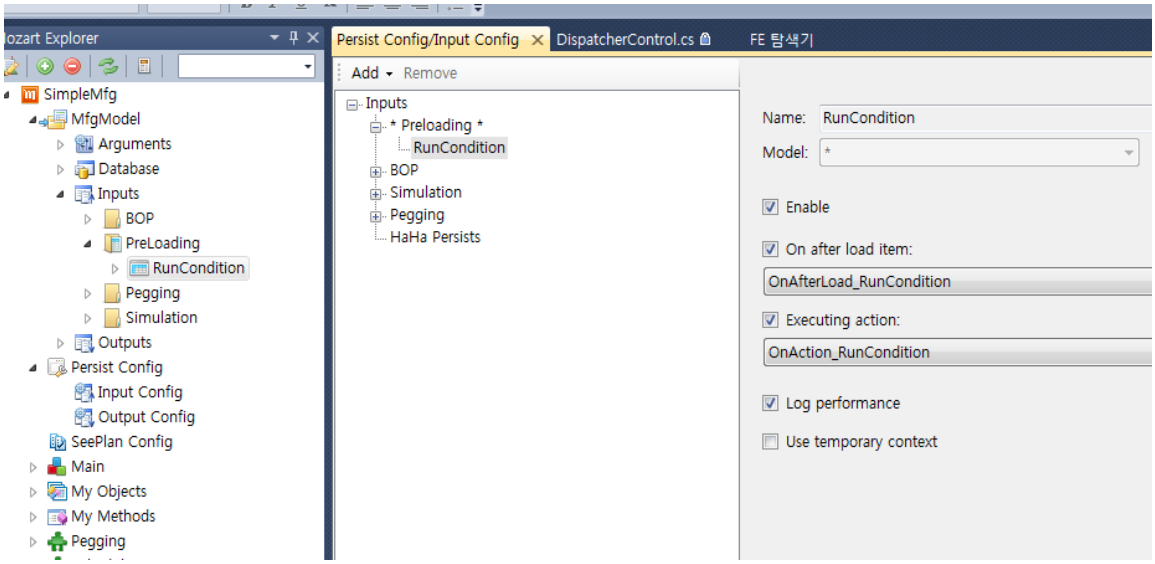
MOZART Main Task 의 실행 구조를 보면 (**Main Control 참조**) 아래의 그림과 같이 Input Data 를 두 번에 거쳐 Download, Loading 하는 단계를 거칩니다.



Pre-loading 과정에서 1차로 Data 를 로딩하고 본 Data 로딩시점에 이 Data 를 사용하여 특정 Data 를 선별적으로 Download 한다던지 특정 Module, 특정 Logic 의 수행여부를 결정하는 등의 코드를 작성할 수 있습니다. 아래는 Pre-loading 설정 방법과 예제입니다.

Preloading 사용방법

1. Input Persist Config 의 "*Preloading* " Group 에 Pre-loading 이 필요한 Input DataItem 을 등록합니다. 아래 그림의 경우 Input DataItem 을 생성하고 해당 Item 을 Pre-loading 하도록 설정한 화면입니다.

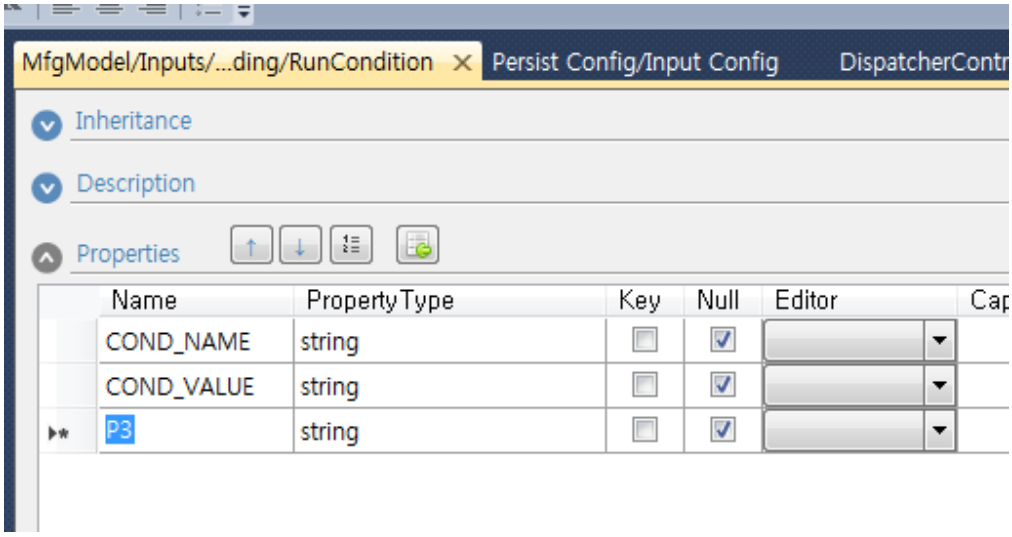


2. Pre-loading 대상 Item 의 경우에도 위 그림과 같이 Persist handler 를 작성할 수 있습니다.

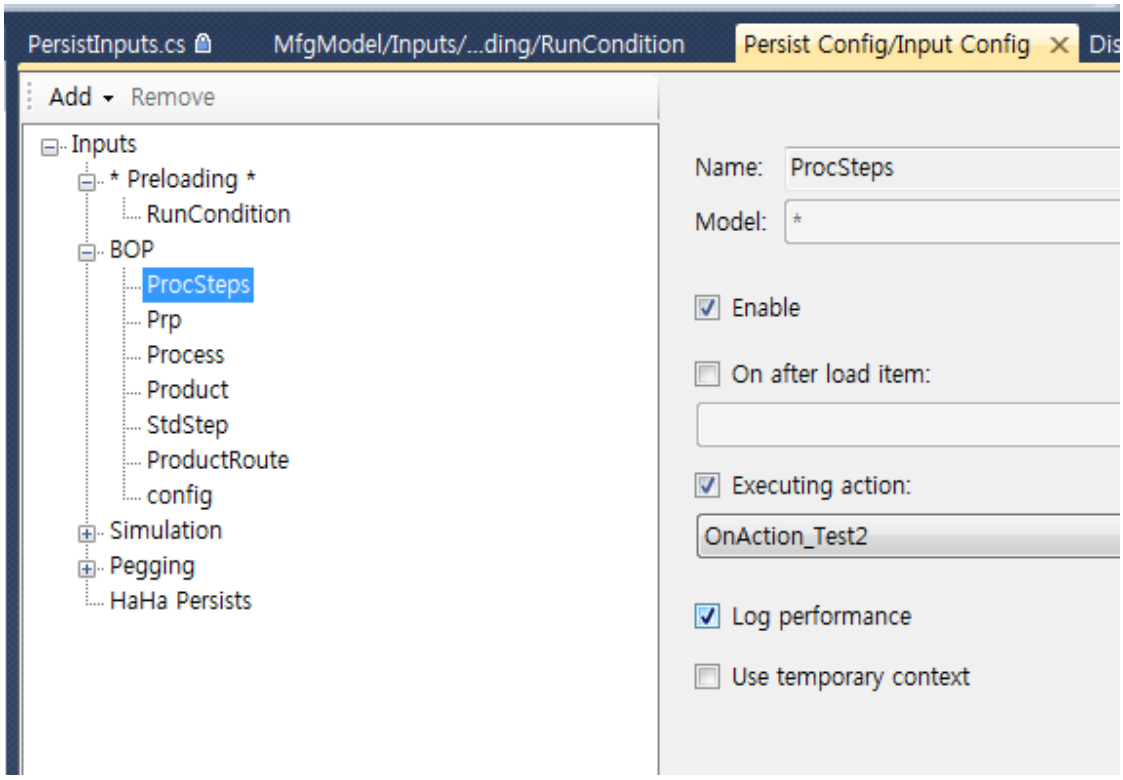
Query Arguments 설정 예제

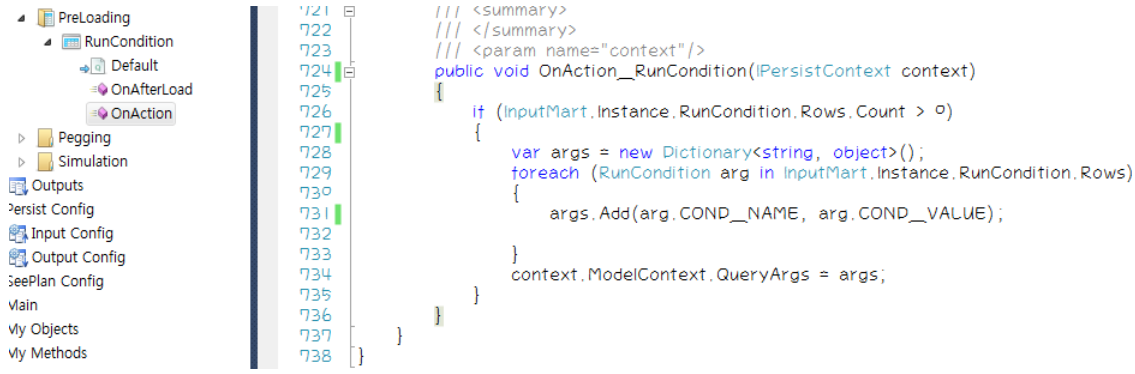
MOZART Task 의 실행 시 사용할 Input Data Query 에 사용되는 Arguments 를 Pre-loading 을 통해 설정하는 예제입니다.

1. 아래 그림과 같이 RunCondition 의 Schema 와 DataAction 을 정의합니다.

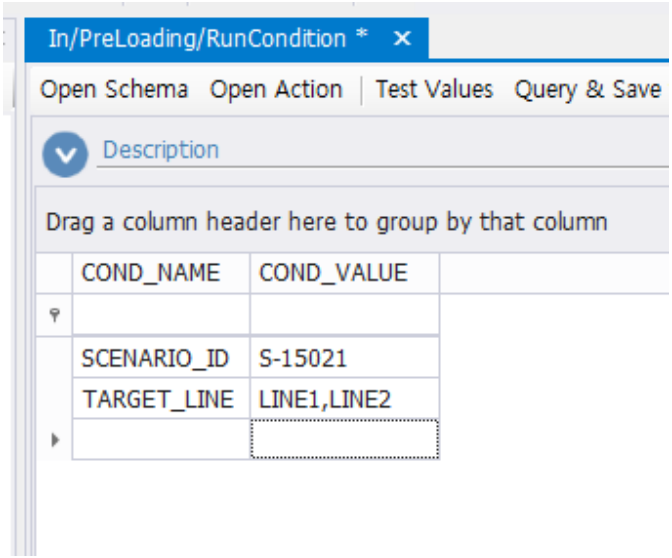


2. RunCondition 을 상기 사용방법과 같이 Preloading Group 으로 옮기고 Handler 를 아래와 같이 작성합니다.





3. RunCondition 의 Data 가 아래와 같이 Download 된 것을 확인 할 수 있습니다.




```
/// <summary>
/// </summary>
/// <param name="context"/>
public void OnAction_Test2(IPersistContext context)
{
    if (context.ModelContext.QueryArgs != null)
    {
        context.ModelContext.QueryArgs                                   Count = 2
        (new System.Collections.Generic.Mscorlib_DictionaryDebugView<string,object>(cor  {[SCENARIO_ID, S-15021]}
        (new System.Collections.Generic.Mscorlib_DictionaryDebugView<string,object>(cor  {[TARGET_LINE, LINE1,LINE2]}
    }

    //    foreach (string a in aaa)
```

4. 본 Data 를 로딩하는 첫 번째 DataItem 인 ProcSteps 의 Action Handler 함수를 확인한 결과 상기 설정 결과가 저장된 모습을 볼 수 있습니다.

SIMULATION MODULE

Simulation 모듈 개요

Mozart SeePlan Library 는 제조업에 특화된 Simulation Project 수행을 위해 Simulation Core Model 의 제어방식을 사용자가 재정의할 수 있는 기능을 제공합니다. 그 외에 제조업 Simulation 의 특성과 Simulation 관점의 확장성을 고려하여 다음의 모델링 기능을 제공합니다. 아래는 Simulation Model 의 기능을 요약합니다.

- **Simulation Model 제어 기능** : Simulation FEComponent 를 통해 FEAction 을 노출하고 구현 Logic (FEDefinition) 을 관리할 수 있습니다.
- **Dispatching Weight Factor Method 기능** : Simulation Library 에서는 제조업의 Disptaching 방식을 지원하기 위해 기본적으로 다중 Weight Factor 에 대한 WeightedSum 방식의 Dispatcher 와 Factor 우선순위에 따른 단계별 Sorting 에 의한 Dispatcher 을 지원합니다. 이 때 사용되는 Weight Factor 에 대한 결과값 가공방식을 사용자가 정의하여 추가하고 이를 Dispatching 시점에 반영할 수 있는 기능을 제공합니다.
- **작업물 기반 통계수집 기능** : 작업물 기반의 통계치 를 정의할 수 있는 기능을 제공합니다.
- **사용자 정의 Event 처리 기능** : Simulation Core 를 통해 발생하는 Event 외에 사용자의 목적에 따라 Simulation Event 를 계획할 수 있고 Event 처리 Logic 을 개발할 수 있는 기능을 제공합니다.
- **Filtering Method 개발기능** : Simulation 중 Dispatching 전 제품/공정/장비에 대한 Filtering 필요 시 Filter Key에 대한 Filtering Set의 각 Filter 조건들을 사용자가 정의하여 추가하고 Filtering 시점에 반영 할 수 있는 기능을 제공합니다.

Simulation Model 의 구동을 위해서는 Model 편집기능과 My Object Module을 활용하여 Simulation In/Output 및 DMO 클래스에 대한 정의가 선행되어야 합니다.

Simulation 모듈 컨트롤 라이브러리

Simulation FEModel 은 Factory, Dispatching, Equipment, Bucketing, SecondResource, Entity, Transfer, JobChangeAgent 의 8개 Category 로 구성되며, 각 Category 별 1~3개의 FEComponent 를 통해 FEAction 을 노출시킵니다. Project 의 Logic 설계자와 개발자는 해당 FEAction 을 통해 Simulation Logic 을 개발할 수 있습니다.

Feature Extension Category 별 역할

- **Factory Category**

  Simulation 객체를 생성하고 객체간 관계를 설정하는 등의 Simulation 초기화를 수행하기 위한 Component 의 집합입니다. Simulation 의 핵심객체인 장비의 초기화를 담당하는 EqpInit, Bucket 초기화를 위한 BucketInit, 기타 Factory 구성 객체의 초기화 및 Factory Event 로 구성됩니다.

- **Entity Category**

  Entity Category 를 구성하는 EntityControl 은 작업물의 생성, 소멸 시점부터 작업물의 이동, 작업진행 과정에서 상태변경이 필요한 모든 시점의 작업물 제어를 담당하며, EntityControl Component 로 구성됩니다.

- **Transfer Category**

  Transfer Category 는 작업물의 이송과 관련된 제어를 수행하며, 기본적인 장비의 이동시간 Model 을 지원하는 TransferControl 과 외부 물류시스템과의 연계 시에 제어를 담당하는 TrasferExtControl 두 가지로 구성됩니다.

- **Dispatching Category**

  Dispatching Control Category 는 1) 작업물이 공정에 도착하는 시점에 작업물 제어를 위한 QueueControl, 2) 작업물의 Dispatching 시작 시점에 장비와 작업물의 조건을 확인하여 Dispatching 대상만을 Filtering 하는 FilterControl, 3) 최종 Dispatching 대상 작업물의 우선순위를 평가하고 선택하는 DispatcherControl 의 세 부분으로 구성됩니다.

- **Equipment Category**

  Equipment Category 는 작업물의 Processing 을 담당하는 주된 Resource 로서 장비에 대한 제어를 담당하는 Component 로 구성됩니다. 장비의 제어는 SetupControl, ProcessControl 및 DownControl 의 3가지 Component 를 통해 수행됩니다.

- **Bucketing Category**

  장비에 작업물을 로딩하는 방식으로 공정을 처리하는 것이 힘든 경우 Buecketing 방식을 사용할 수 있습니다. 이러한 Bucketing Logic 제어를 위한 Component 는

BucketControl 하나로 구성됩니다.

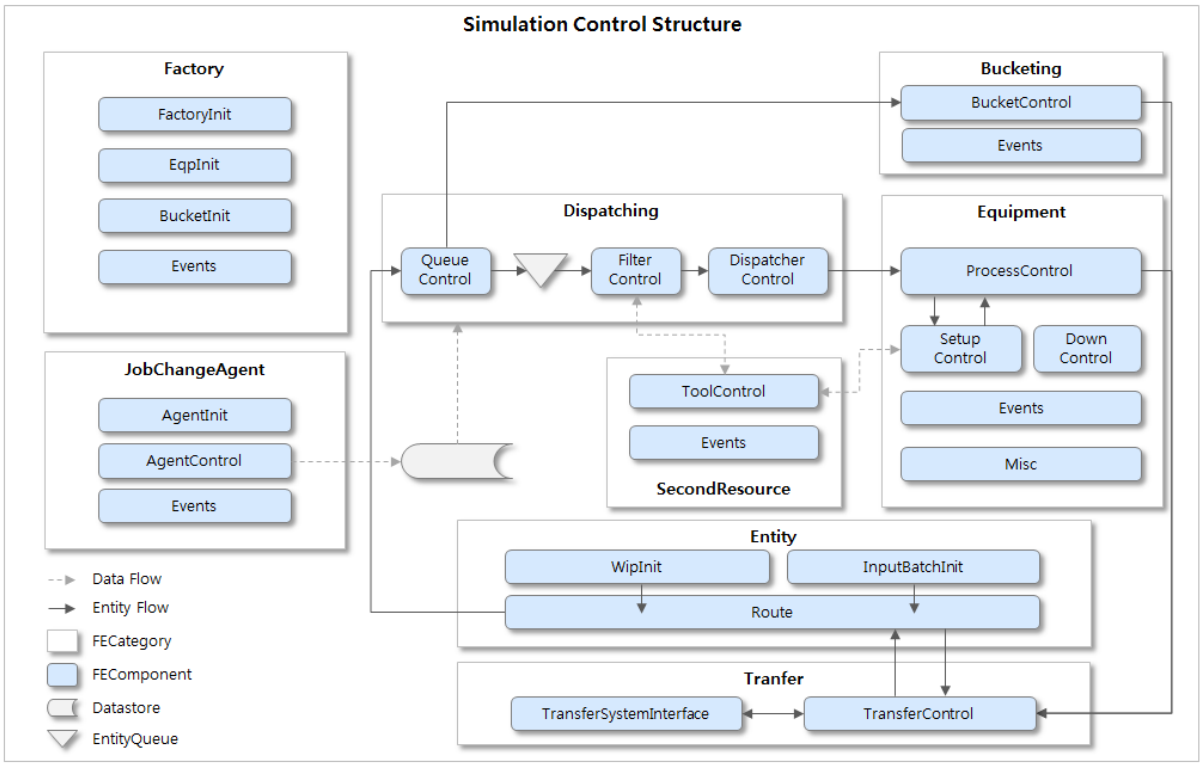
- **SecondResource Category**

  공정수행 시 장비와 작업물에 의해 점유되고 사용되는 Second Resource 에 대한 제어 Model 을 포함합니다. SecondResource 의 경우 SetupCrew 와 같은 능동형 2차 자원과 작업물, 장비에 장착되는 Tool, Jig 등의 수동형 자원에 대한 제어를 모두 포함하는 SecondResourceControl 로 구성됩니다.

- **JobChangeAgent Category**

  Agent 는 일정 주기로 작업물의 투입 및 장비의 작업교체, 라인 간 적정물량의 분배 (Balancing) 과 같은 특정 목적을 위한 Data Logic Processing 을 수행하도록 정의된 제어 Model 입니다. SeePlan Library 에서는 JobChangeControl, InputControl Component 를 제공합니다.

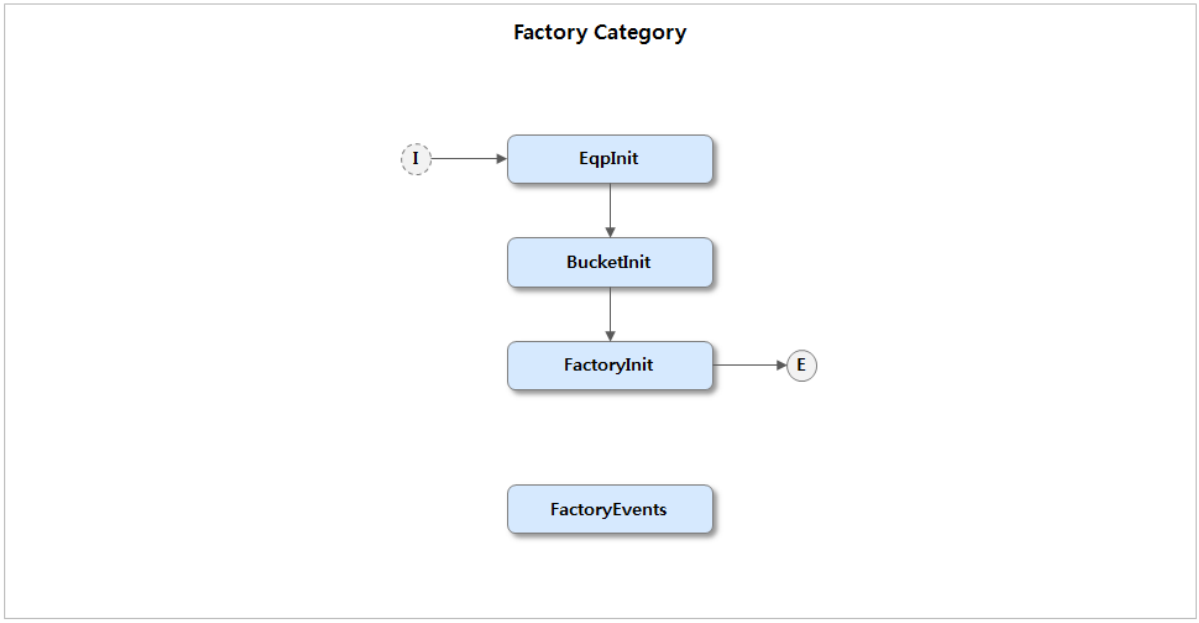
아래 그림은 각 FECategory 와 FEComponent 간의 관계를 표시합니다.



Factory

Simulation 객체를 생성하고 객체간 관계를 설정하는 등의 Simulation 초기화를 수행하기 위한 Component 의 집합입니다. 초기화 부분 중에 Resource 와 관련된 초기화를 담당합니다.

구성요소는 다음의 네 가지 Component 입니다. Factory Category 의 Component 호출 순서는 일반적으로 아래의 순서를 따릅니다.



EqpInit

장비 (Equipment) 와 Dispatcher, DispatchingAgent 를 생성하고 초기화합니다.

BucketInit

Capacity Bucket 및 각종 제약을 위해 필요한 Bucket 을 생성하고 초기화합니다.

FactoryInit

Dispatching Preset 사용을 위한 Factor Method 의 등록 및 통계수집을 위한 Monitor 를 등록합니다.

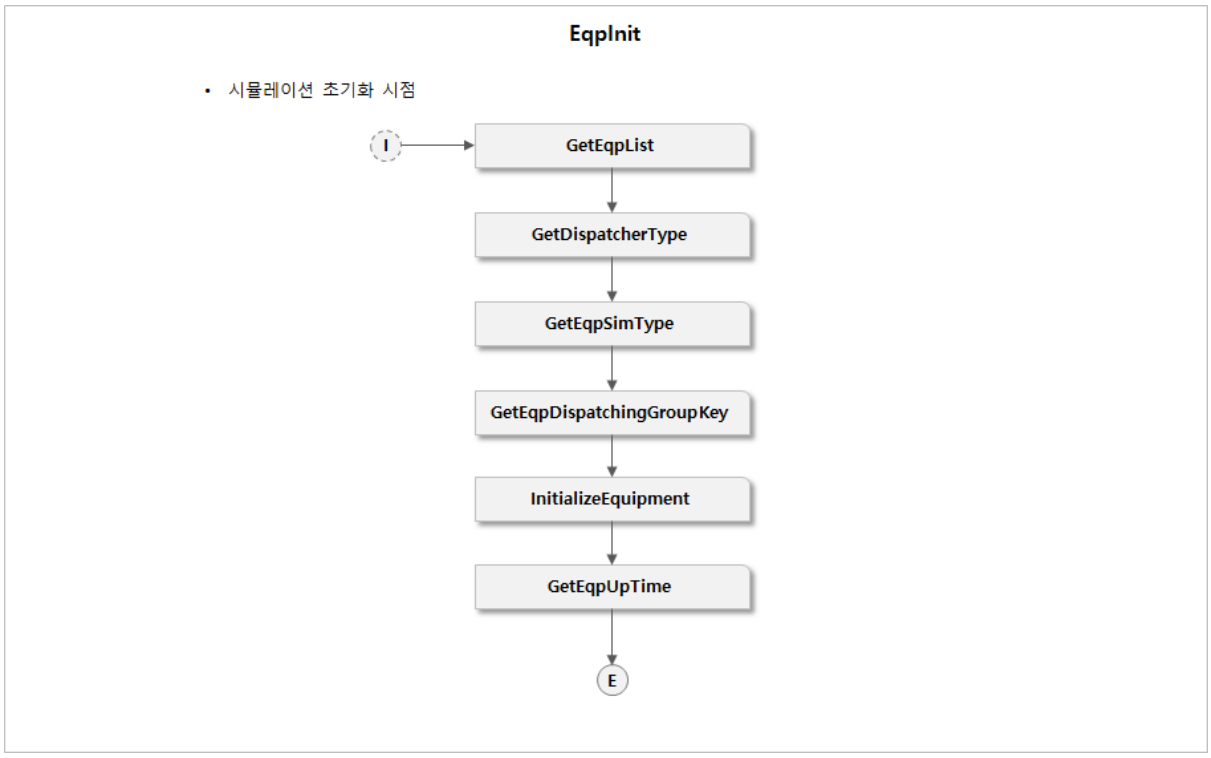
FactoryEvents

Simulation 전체 Model (Factory Model) 의 개별 Model 들에 영향을 미치는 Factory 의 Event 처리 함수를 노출시킵니다.

EqpInit 컨트롤

Simulation 에서 사용될 장비를 생성하고 장비의 Simulation 을 위한 장비 Simulation Reference Model 및 Dispatching 방식 등을 설정하도록 구성된 FEComponent 입니다. 장비 (*AoEquipment*) 를 생성한 후 공장 통합 Model (*AoFactory*) 에 등록하는 작업을 수행합니다.

다음은 EqpInit 의 FEAction 간 상관관계 및 호출 시점을 도식화한 그림입니다.



EqpInit 은 아래의 네 가지 Action 을 통해 장비 초기화를 위한 사용자 정의 Logic 을 추가할 수 있도록 합니다.

1. ***GetEqpList(*)*** : Simulation 대상이 되는 장비 (Resource) List 를 반환하는 Logic 을 구현합니다. 대상 장비 List 에 포함된 장비만 Simulation 장비로 생성이 됩니다. 본 Action 은 Simulation 을 위한 필수 구현 Action 입니다. 다만, Bucketing 만을 통해 Foward Simulation 을 수행할 경우에는 구현하지 않아도 됩니다.

2. ***GetDispatcherType*** : 장비별로 사용하는 Dispatcher Type을 지정합니다. SeePlan 에서 Weight Preset 정보에 기반한 WeightSumDispatcher 와 WeightSortedDispatcher 를 제공하며, Project 의 장비 Data 에 장비가 사용할 Dispatcher 를 설정하여 로딩하여 초기화 시점에 지정할 수 있습니다.

   Default code sample

   ```
   public string GET_DISPATCHER_TYPE_DEF(Mozart.SeePlan.DataModel.Resource eqp, ref boo
   l handled, string prevReturnValue)
   ```

```
{
    return eqp.DispatcherType.ToString();
}
```

3. ***GetEqpSimType*** : 장비가 사용할 Simulation 장비 타입을 설정하도록 합니다. 기본적으로 SimType 은 Input Schema 를 통해 장비의 DataModel 에 로드하는 기본 Data 이지만, 경우에 따라 기본 SimType 을 사용자 Logic 에 따라 변경해서 사용할 필요가 있을 수 있습니다.

   Default Logic Example

```
public string GET_EQP_SIM_TYPE_DEF(Mozart.SeePlan.DataModel.Resource eqp, ref bool h
andled, string prevReturnValue)
{
   return eqp.SimType.ToString();
}
```

4. ***GetEqpDispatchingGroupKey*** : 장비의 특정 조건에 따라 Schedule Group 이 결정되고 해당 장비 Group 내에서만 작업물이 가공되는 경우 DispatchingAgent 를 별도로 생성하여 Dispatching 하도록 모델링 할 수 있습니다. 이 경우 장비별 DispatchingAgent 를 Mapping 하는 Logic 이 필요하며 본 FEAction 을 통해 해당 Logic 을 구현합니다. 단일 DispatchingAgent 를 사용하여 장비의 Dispatching 을 처리하는 경우에는 해당 Action 을 구현하지 않아도 됩니다.

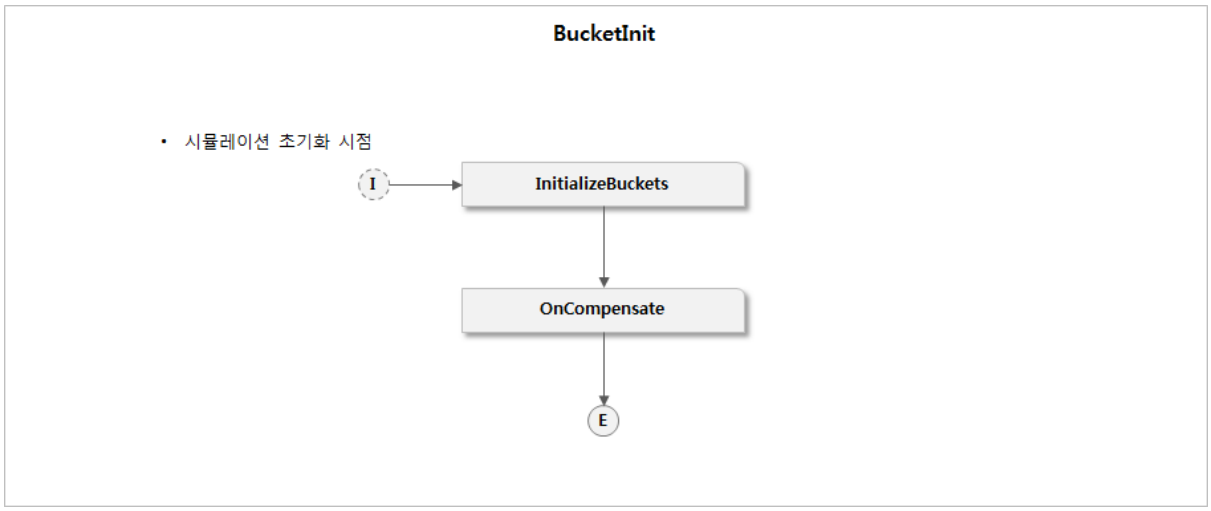
5. ***InitializeEquipment*** : 장비(AoEquipment) 가 생성된 직후에 장비의 속성을 초기화하기 위해 사용합니다. SimEqptype = Inline 장비의 경우 작업물 한 Unit 이 장비에서 Processing start ~ end 할때까지의 소요시간인 flow time 을 사용하여 event schedule 할지 작업물(Lot) 의 작업시작~종료까지의 시간인 process time 을 쓸지 여부를 설정하는 등의 작업을 할 수 있습니다. 이 속성은 ***UseProcessingTime*** 속성입니다. 기본값은 true 이며, true 인 경우 processing time 을 사용합니다. 사용자 조직에서 작업시간 집계를 어느 단위로 하는지에 따라 설정을 바꿔서 사용할 수 있습니다.

6. ***GetEqpUpTime*** : 장비의 상태가 초기 Down 상태인 경우에 Simulation 종료시각 이전에 UP 상태로 변경이 필요한 경우 본 함수에서 장비 UP 시각을 설정합니다.

BucketInit 컨트롤

Simulation 모델링 시 제조 Resource 를 장비단위로 모델링 하지 않고, Bucketing 을 통해 처리하는 경우에 Bucket Capacity 를 결정할 수 있는 Capacity Bucket 을 생성합니다. 제품/공

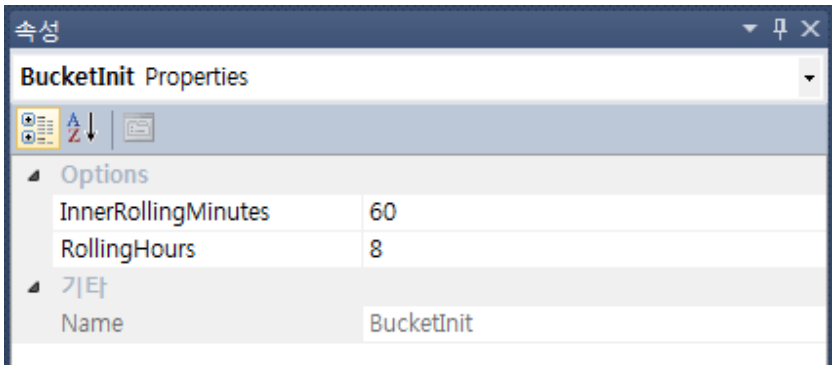
정별로 Bucketing 수행 시 적용할 수 있는 Constraint 에 대해서도 필요 시 Bucket Key 에 따라 Bucket 을 정의하여 추가하는 작업을 수행할 수 있습니다.



1. ***InitializeBuckets*** : Bucketing 시 사용될 장비 Group (공정 Group) 에 대한 Capacity Bucket 를 생성하고 속성을 설정합니다. 또한 필요 시, 특정 Bucket 에 대한 제품(공정)제약을 Bucket 형식으로 생성할 수 있습니다.
2. ***OnCompensate*** : Bucket Rolling 의 기준 시간은 Factory Start Time 으로 만일 Simulation 이 Bucket Rolling 주기의 중간에 시작되는 경우에는 Factory Start Time 과 Rolling 시간을 고려하여 Bucket 의 Capacity를 현실적으로 보정하는 작업을 수행해야 합니다. Library 에서 제공하는 기본 Logic 은 Capacity Bucket 의 경우 시간 비율만큼 실적이 발생한 것으로 보고 초기화를 수행합니다. 사용자 정의 방식을 사용하고자 할 때는 본 Action 을 구현해주어야 합니다.

Bucketing 속성설정

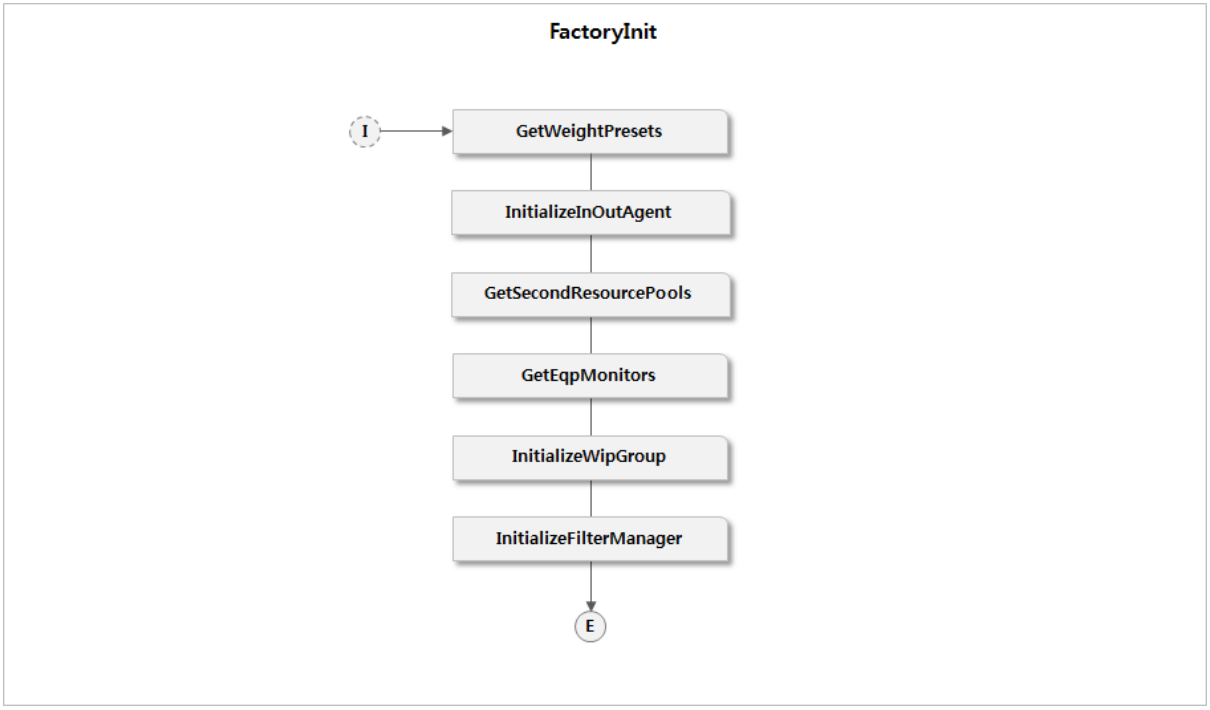
Bucketing 을 위한 Rolling 주기 속성은 아래 그림과 같이 Visual Studio 의 속성창에서 설정할 수 있습니다. 설정 가능한 속성은 두 가지 입니다.



- **InnerRollingMinutes** : 실제 Bucket Rolling 이 발생하는 주기를 정의합니다. 분 단위로 설정합니다. 기본값은 60분으로, 1시간 단위로 Rolling 합니다.
- **RollingHour** : Capacity 설정 기간을 시간단위로 설정합니다. 상기 그림에서와 같이 8시간으로 설정한 경우, Bucket 의 Capacity 설정 시 8시간 단위로 설정해 주어야 합니다.

Factory Init 컨트롤

Factory 를 구성하는 자원 및 Simulation 결과집계 등과 관련된 요소들의 초기화를 담당하는 Component 입니다.



1. ***GetWeightPresets*** : MOZART SEEPLAN Library 의 경우 기본 Dispatcher 를 제공합니다. 이 중 Fifo Dispatcher 를 제외하고 나머지 WeightSumDispatcher 와 WeightSortedDispatcher 의 경우 모두 Preset 를 사용합니다. 사용자는 이 Preset 기반의 Dispatcher 를 사용할 경우 해당 Preset 에서 사용하는 Factor 별 평가함수를 Simulation Model 상의 Weights 기능을 통해 구현해야 합니다. 또한 각 장비 Group (혹은 장비) 별로 Dispatching 시 사용할 Preset 정보를 설정해야 합니다. 본 Action 은 사전에 정의된 Preset 전체 List 정보를 반환하도록 구현하여 Dispatcher 를 초기화 합니다. Preset 기반이 아닌 Fifo 방식의 Dispatcher 를 사용하는 경우에는 구현하지 않습니다.
2. ***InitializeInOutAgent*** : 라인 혹은 특정 공정구간의 투입시점에 투입제약을 판별하여 작업물의 투입을 결정하기 위한 Agent 컨트롤을 생성하고 추가합니다. Library 의

InOutAgent 클래스를 사용하여 객체를 생성하고 실행조건을 설정합니다.

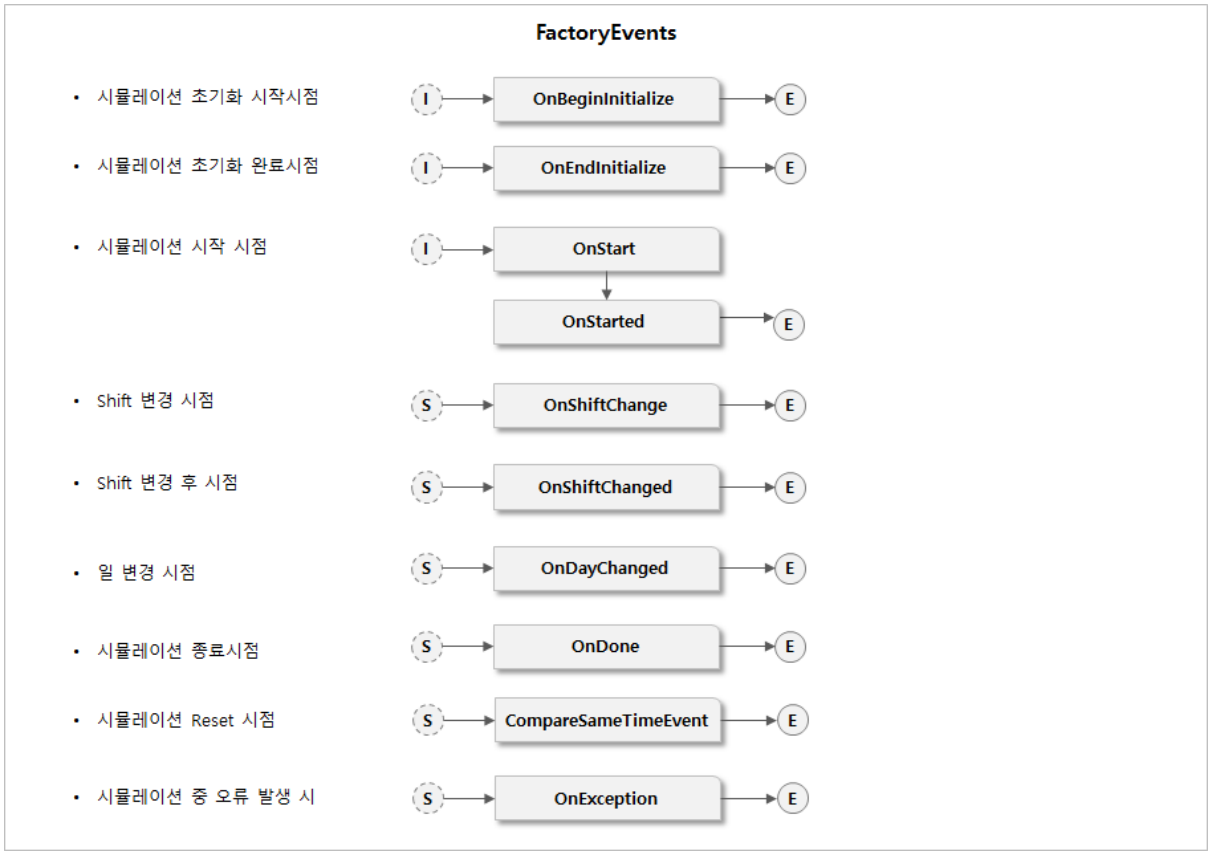
코드 작성 예제

```
// 생산 라인의 투입을 제어하기 위해 InputAgent 를 설정합니다.
// Agent 의 투입 실행 주기는 1시간으로 설정합니다.
// Agent 의 Key 는 "InputControl" 이라는 이름을 생성합니다.
public IList<InOutAgent> INITIALIZE_IN_OUT_AGENTS0(AoFactory factory, ref bool handl
ed, IList<InOutAgent> prevReturnValue)
{
    List<InOutAgent> agents = new List<InOutAgent>();
    InOutAgent InAgent = new InOutAgent(factory, "InputControl") ;
    InAgent.Duration = Time.FromHours(1);
    agents.Add(InAgent) ;
    return agents ;
}
```

3. ***GetSecondResourcePools*** : Simulation 에서 사용하게 되는 각 SecondResource 와 SecondResourcePool 을 생성하여 반환합니다.
4. ***GetEqpMonitors*** : Simulation 장비 통계치를 수집할 IEquipmentMonitor 구현 클래스를 반환합니다. 개별적으로 구현할 수 있으며, Domain Library 에서 제공하는 Definition 을 조회하여 Binding 할 수 있습니다.
5. ***InitializeWipGroup*** : Simulation 중 여러 목적에 따라 분류된 Wip 통계치를 사용하기 위해 Group 을 정의합니다.
6. ***InitializeFilterManager*** : Simulation 중 Dispatching 전 작업물에 대한 Filtering 진행을 위하여 Filter Key 및 Key 해당하는 Filter List 정보를 설정합니다. Filtering 조건들은 Simulation Model상의 Filters 기능을 통해 구현해야합니다. 이후 **Dispatching > FilterControl >GetFilterSetKey**를 통해 Filter Key를 반환하면 **DoFilter** 함수에서 Key 해당하는 Filter Set을 호출하여 Filters에 구현된 조건대로 Filtering을 진행합니다.

Factory Events

Plan & Schedule Module Reference > Simulation Module > Simulation Control Library > Factory > Factory Events Simulation 전체 객체에 영향을 미치는 Event 에 대한 처리 함수를 사용자 정의할 수 있는 Component 입니다.

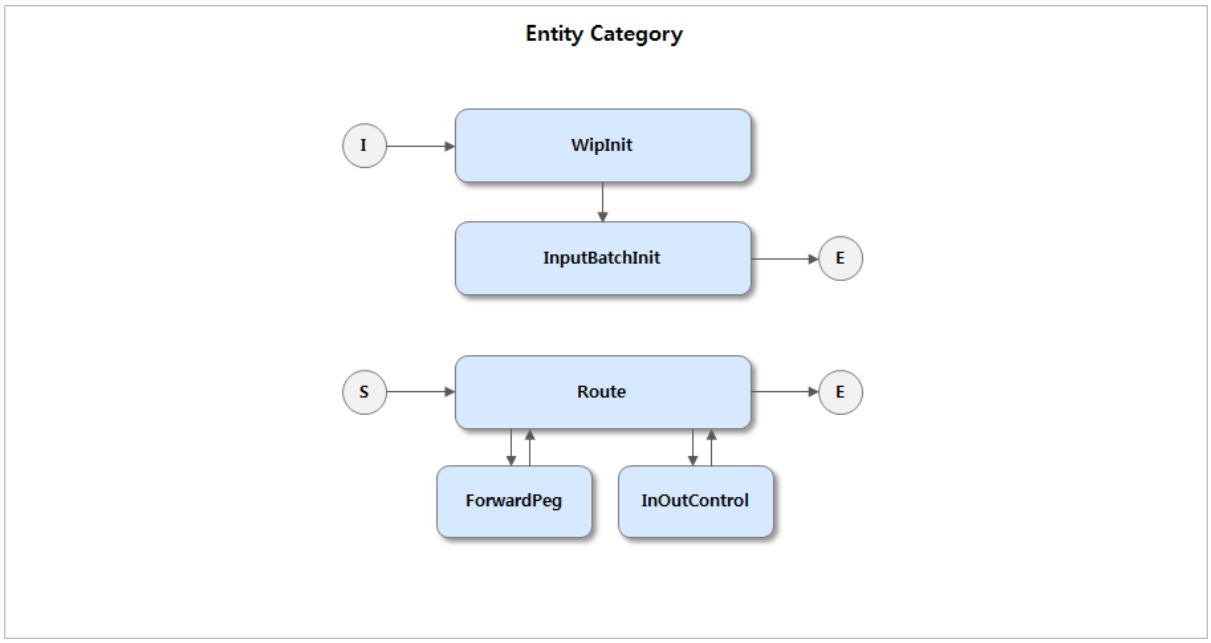


1. ***OnBeginInitialize*** : Simulation 수행을 위한 초기화 시작 시점에 호출되는 함수입니다.
2. ***OnEndInitialize*** : Simulation 초기화 과정이 모두 종료된 시점에 호출되는 함수입니다.
3. ***OnStart*** : Simualtion 전체 Model 인 Factory Object 가 생성되고 Simulation 이 시작되는 시점에 호출됩니다. Simulation 시작시점에 초기화가 필요한 사용자 정의 로직을 구현할 수 있습니다.
4. ***OnStarted*** : Simulation 이 시작된 직후에 호출됩니다. 모든 시뮬레이션 초기화가 끝나고 수행되어야 할 로직을 구현할 수 있습니다.
5. ***OnShiftChange*** : Simulation 중 설정된 Shift 가 변경되는 시점에 호출되는 함수입니다. Shift 변경 시점에서 가장 먼저 호출되며, 이 Action 이 호출된 뒤에 다른 작업이 시작됩니다.
6. ***OnShiftChanged*** : Simulation 중 설정된 Shift 가 변경된 뒤에 호출되는 함수입니다. Shift 변경 시점에서 실행되는 모든 작업이 끝난 뒤에 호출됩니다. 일반적인 제조 공장의 경우 작업 교대조가 변경되는 시점에 실적의 마감, 가동율 등 통계 수치 조정, 제약의 초기화 등의 업무를 처리하는 경우가 많습니다. Simulation 에서 이러한 실제 Model 의 Behavior 를 모델링하고자 할 경우 본 Action 에 해당 Logic 을 구현합니다.

7. ***OnDayChanged*** : OnShiftChanged 와 유사하게 공장의 일 변경시점에 처리되어야 하는 각종 정보의 업데이트 등 Logic 을 처리합니다.
8. ***OnDone*** : Simulation 의 Root Active Object 인 Factory 의 종료시점에 호출되는 함수입니다. Simulation 에서 PlanPeriod 의 마지막 이벤트가 발생한 후 호출됩니다. 이때 처리할 Logic 을 구현합니다. 참고로 Mozart Simulation 모듈에서는 ModelContext.EndTime 에 시뮬레이션 종료 이벤트를 발생시키지 않기 때문에 OnDone 의 currentTime이 Simulation EndTime과 다를 수 있습니다.
9. ***CompareSameTimeEvent***: Simulation이 다시 초기화되는 시점에 동시에 발생하는 이벤트가 여러개 존재하는 경우 이벤트간 우선선위를 설정하는 로직을 구현 할 수 있는 FEAction입니다. 일반적으로는 설비 관련 이벤트간 우선순위를 설정하기 위해 사용합니다.
10. ***OnException***: Simulation 수행 중 자주 발생하는 라이브러리 예외처리를 사용자가 제어할 수 있는 FEAction입니다. 본 FEAction을 통해 예외처리를 기록하거나 예외처리가 발생 후 Simulation 수행 지속 여부 등을 처리 할 수 있습니다. `OnException` FEAction이 호출되는 Simulation Exception 목록 및 동작방식은 여기를 통해 확인 바랍니다.

Entity

Entity Category 는 Simulation Entity 를 생성하는 두 가지 Component 와 Simulation 전체 주기동안 작업물의 Routing 과 관련된 모든 Logic 을 담당하는 Route Component 로 구성됩니다.



WipInit

WipInit 은 재공의 상태와 Simulation 의 작업물 초기상태를 동기화 시켜주는 역할을 수행하며, Simulation 초기 재공을 생성하고 초기화 합니다.

InputBatchInit

InputBatchInit 은 투입계획이나 투입 Rule 에 따라 라인에 투입할 신규 작업물을 생성하고, 투입시점을 결정하는 일련의 작업을 처리하는 FEComponent 입니다.

Route

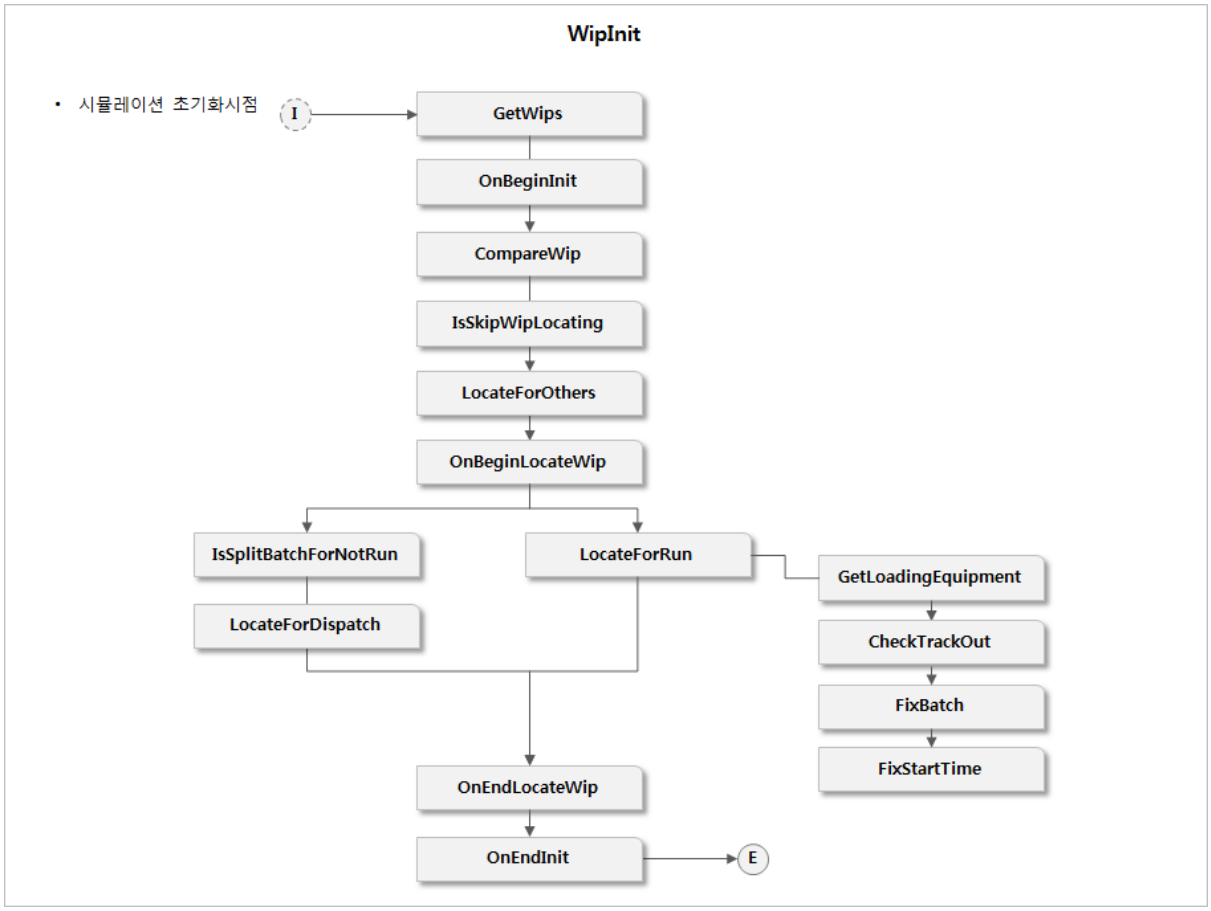
작업물의 생성 이후부터 소멸시점까지 작업물의 공정 흐름 제어, 작업물 상태 변경 시점의 Data 처리 방식을 제어합니다. Route 는 작업물을 매개로 Simulation 을 구성하는 Dispatching Control, Resource Control 영역의 FEComponent 와 상호 연관 관계를 형성합니다.

ForwardPeg

Simulation 중 작업물의 생산 목표 제품을 변경하면서 진행하는 경우, 생산목표 중 작업물의 목표가 되는 목표를 지정하는 작업을 수행하기 위한 Logic 을 담당하는 FEComponent 입니다.

WipInit 컨트롤

WipInit은 재공의 상태와 Simulation 의 작업물 초기상태를 동기화시켜주는 역할을 수행하며, 이와 관련된 일련의 Logic 을 사용자가 개발할 수 있도록 지원하는 FEComponent 입니다. 재공은 상태에 따라 이동/대기/가공중 재공으로 분류할 수 있으며, 대기 상태의 경우 장비에 로딩되어있는 경우와 장비 외부에 대기중인 상태로 나뉠 수 있습니다.



1. **GetWips** : 재공상태의 작업물 List 를 반환하는 Logic 을 구현합니다. Simulation 의 수행을 위해 반드시 구현되어야 하는 필수 FEAction 입니다.
2. ***OnBeginInit*** : 초기화 대상 Lot 을 받아온 후 실제 초기화를 진행하기 직전에 호출되는 FEAction 입니다.
3. ***CompareWip*** : 초기화될 재공에 대해 상태와 시간정보를 기준으로 초기화의 순서를 결정할 수 있도록 구현합니다. 이동, 대기중인 작업물의 경우에는 초기화 순서가 큰 영향을 미치지 않으나 장비에서 대기중이거나, 작업중인 작업물의 경우 초기화 순서에 따라 초기화에 문제가 될 수 있습니다.

   Default Logic Example

```
public int COMPARE_WIP_DEF(IHandlingBatch x, IHandlingBatch y, ref bool handled, int
prevReturnValue)
{
    if (object.ReferenceEquals(x, y))
        return 0;

    var a = (ILot)x.Sample;
    var b = (ILot)y.Sample;
```

```
    // RUN-INBUF->WAIT->HOLD
    int cmp = b.CurrentState.CompareTo(a.CurrentState);
    if (cmp != 0)
        return cmp;

    return cmp;
}
```

4. ***IsSkipWipLocating*** : 작업물에 대한 초기화 정보가 불충분하거나 Data 처리 오류 있다고 판단되는 경우 작업물은 초기화하지 않도록 Filtering 하도록 Logic 을 구현합니다. 본 Action 은 개별 작업물 단위로 처리되며, 작업물의 초기화를 수행하지 않는 경우 순서에 따라 다음 작업물에 대한 초기화 진행여부를 판별합니다.
5. ***IsSplitBatchForNotRun*** : 배치 작업물이 Run 상태가 아닌 경우 개별 Lot 으로 Split 해서 처리할 것인지 여부를 결정합니다. 기본값은 true 로 Run 상태가 아닌 배치 작업물은 단위 작업물로 분리되어 초기화됩니다. false 인 경우에는 배치를 유지한 상태로 초기화 됩니다.
6. ***LocateForOths*** : 초기 재공을 초기화할 때 기본 상태가 아닌 Hold 상태의 재공을 처리하거나, 재공의 현 공정이 Bucketing 을 통해 처리되어야 하는 경우 및 그 외 정상적으로 대기, 이동, 장비가공 상태로 초기화할 수 없는 작업물을 초기화 처리하는 Logic 을 구현합니다. 이 경우 해당 작업물은 사용자에 의해 별도 Logic 으로 처리되어야 합니다.
7. ***OnBeginLocateWip*** : 작업물이 정상적인 상태인 경우 상태별 초기화하기 시작하는 시점에 호출되며, 이 때 필요한 사용자 정의 Logic 을 구현합니다. 상태가 가공중인 경우 작업물에 필요한 Second Resource 를 설정하나 기타 작업물의 속성을 설정하는 용도로 사용합니다.
8. ***LocateForDispatch*** : 작업물을 대기상태로 처리하는 Logic 을 구현합니다.

   Default Logic Example

```
public void LOCATE_FOR_DISPATCH_DEF(AoFactory factory, IHandlingBatch hb, ref bool h
andled)
{
    if (hb.IsFinished)
    {
        factory.Router.AddInitial((Entity)hb, hb.IsFinished);
    }
    else
    {
        var router = EntityControl.Instance;

        string dispatchKey = router.GetLotDispatchingKey(hb);
        DispatchingAgent da = factory.GetDispatchingAgent(dispatchKey);
        if (da == null)
            throw new InvalidOperationException("DispatchingAgent is not registerd,
 check 'Route/GetDispatchingKey' implementation");
        da.Take(hb);
```

```
    }
}
```

9. ***LocateForRun*** : 재공의 초기상태가 Run 인 경우를 처리하는 Logic 을 구현합니다. 기본 Logic 은 Library 에서 제공하며, 이 때 ***GetLoadingEquipment, CheckTrackOut*** Action 을 사용합니다. 만일 별도로 본 Action 의 Definition 을 추가하는 경우 두 Action의 구현내용은 사용되지 않습니다.

   Default Logic Example

```
public void LOCATE_FOR_RUN_DEF(AoFactory factory, IHandlingBatch hb, ref bool handle
d)
{
    var wipInitiator = ServiceLocator.Resolve<WipInitiator>();
    AoEquipment eqp = null;
    string eqpID = wipInitiator.GetLoadingEquipment(hb);
    // 재공이 Run 상태이나 실제 Processing 이 끝난 상태로 Outport 에 위치하는지 여부 판별
    bool trackOut = wipInitiator.CheckTrackOut(factory, hb);
    if (string.IsNullOrEmpty(eqpID) || factory.Equipments.TryGetValue(eqpID, out eq
p) == false)
    {
        //장비가 없으면 Bucketing 처리
        factory.AddToBucketer(hb);
        Logger.Warn("Eqp {0} is invalid, so locate running wip to dummy bucket. chec
k input data!", eqpID ?? "-");
    }
    else
    {
        if (trackOut)
        {
            eqp.AddOutBuffer(hb);
        }
        else
        {
            eqp.AddRun(hb);
        }
    }
}
```

10. ***GetLoadingEquipment*** : 작업물이 현재 로딩중인 경우 로딩중인 장비의 ID 를 반환하는 Logic 을 구현합니다. 재공의 초기 상태가 장비에 로딩되어 대기중인 경우나 가공중인 경우 반드시 구현이 필요한 Action 입니다. 본 함수는 LocateForRun 함수가 재정의되는 경우 호출되지 않습니다.

11. ***CheckTrackOut*** : 작업물 상태는 Processing 이지만 작업물의 작업시작 정보를 기준으로 이미 작업이 끝났는지 판단하는 Logic 을 구현합니다. LocateForRun 함수가 재정의되는 경우 호출되지 않습니다.

12. ***FixBatch*** : 초기 Run Wip 의 상태가 장비의 가공타입 (Batch) 과 맞지 않는 경우 이를 처리합니다. 장비는 Batch Type 인데 작업물이 개별 형태이거나 혹은 그 반대의 경우가 이

에 해당합니다. 이럴 경우 시스템에서 처리할 수 있는 유효한 Data 로 만들어 초기화 할 수 있도록 Logic 을 처리합니다. LocateForRun 함수가 재정의되는 경우 호출되지 않습니다.

Default Logic Example

```
public virtual ISimEntity FIX_BATCH_DEF(AoEquipment aeqp, ISimEntity entity, ref boo
l handled, ISimEntity prevReturnValue)
{
    if (aeqp.IsBatchType() && (entity is BatchEntity) == false)
    {
        BatchEntity lbatch = new BatchEntity();
        lbatch.Contents.Add(entity);
        entity = lbatch;
    }
    return entity;
}
```

13. ***FixStartTime*** : 초기 Run Wip 의 시작시각 정보가 정합하지 않은 경우 이를 처리합니다. 작업물의 가공 시작 시간과 현재 시각 (Simulation 시작시각) 의 차가 작업물의 현공정 가공시간 및 정상 가공시간의 범위를 벗어나는 경우에는 시작시각 혹은 상태정보를 비정상으로 보고 작업물과 장비의 초기 상태를 Rule 에 의해 재정의하도록 구현합니다. LocateForRun 함수가 재정의되는 경우 호출되지 않습니다.

    Default Logic Example

```
public virtual DateTime FIX_START_TIME_DEF(AoEquipment aeqp, IHandlingBatch hb, ref
 bool handled, DateTime prevReturnValue)
{
    return aeqp.NowDT;
}
```

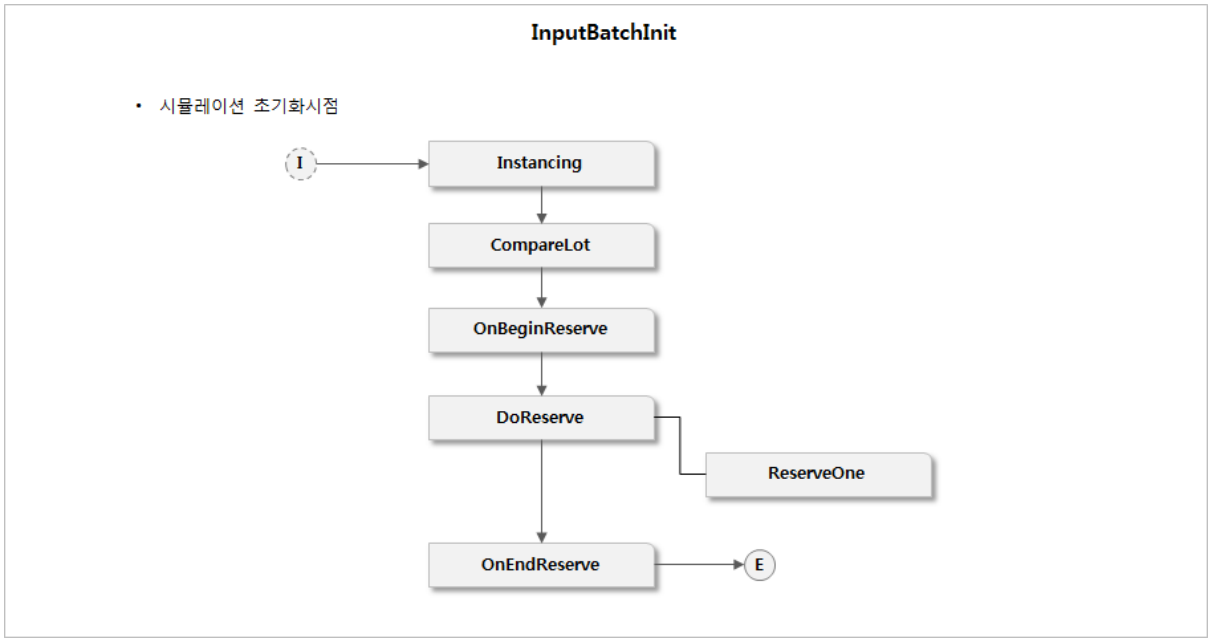
14. ***OnEndLocateWip*** : 개별 재공에 대한 처리가 끝난 후에 호출됩니다.
15. ***OnEndInit*** : 모든 재공초기화가 끝난 후에 호출됩니다.

InputBatchInit 컨트롤

InputBatchInit 은 투입계획이나 투입 Rule 에 따라 라인에 투입할 신규 작업물을 생성하고, 투입시점을 결정하는 일련의 작업을 처리하는 FEComponent 입니다. 일반적으로 작업물은 투입계획에 따라 생성되고, 투입공정의 Bottleneck 장비 Capacity, 투입 Volume, 투입 Batch size 등을 고려하여 투입량, 투입시점을 결정합니다. 별도의 InputControl 을 사용하여 비교적

짧은 주기로 투입을 Lot 을 결정하는 경우나 Simulation 특성상 추가적인 Lot 투입이 필요없는 경우에는 본 Control 에 대한 구현을 하지 않습니다.

아래는 InputBatchInit 의 Action 구성 및 순서를 표시합니다.



1. ***Instancing*** : 투입대상 작업물을 생성합니다. BatchInitiator를 사용하는 경우에는 일반적으로 제품별 일별 투입 Target 이나 투입계획이 Input이 됩니다.
2. ***CompareLot*** : Instancing 에서 생성된 Lot 정보를 투입 순서에 따라 Sorting 합니다.

   Default Logic Example

   ```
   public int COMPARE_LOT_DEF(ILot x, ILot y, ref bool handled, int prevReturnValue)
   {
       if (Object.ReferenceEquals(x, y))
           return 0;
       return x.ReleaseTime.CompareTo(y.ReleaseTime);
   }
   ```

3. ***OnBeginReserve*** : 투입대상 Lot에 대해 투입 대상으로 등록하기 전 사전 처리가 필요한 Logic 을 구현합니다.
4. ***DoReserve*** : 투입 대상 작업물 List 를 Factory.BatchRelease 에 투입예약 등록합니다. Factory.BatchRelease 는 등록된 Lot을 예약시점에 라인에 투입하는 역할을 수행합니다. 본 Action 은 Library 에 기본 구현되어 있으며, ReserveOne Action 을 호출하게 됩니다. 따라서 본 Action 에 별도의 Definition 을 하는 경우 ReserveOne Action 의 구현은 사용되지 않습니다.

Default Logic Example

```
public void DO_RESERVE_DEF(List<ILot> lots, ref bool handled)
{
    var batchInitiator = ServiceLocator.Resolve<BatchInitiator>();
    int index = 0;
    while (index < lots.Count)
    {
        batchInitiator.ReserveOne(lots, ref index);
    }
}
```

5. ***ReserveOne*** : 투입 대상 작업물을 개별적으로 투입예약 등록할 때 작업물별로 처리해야 하는 Logic 을 구현합니다. 본 Action 은 DoReserve 를 재정의하는 경우 사용되지 않습니다.

   Default Logic Example

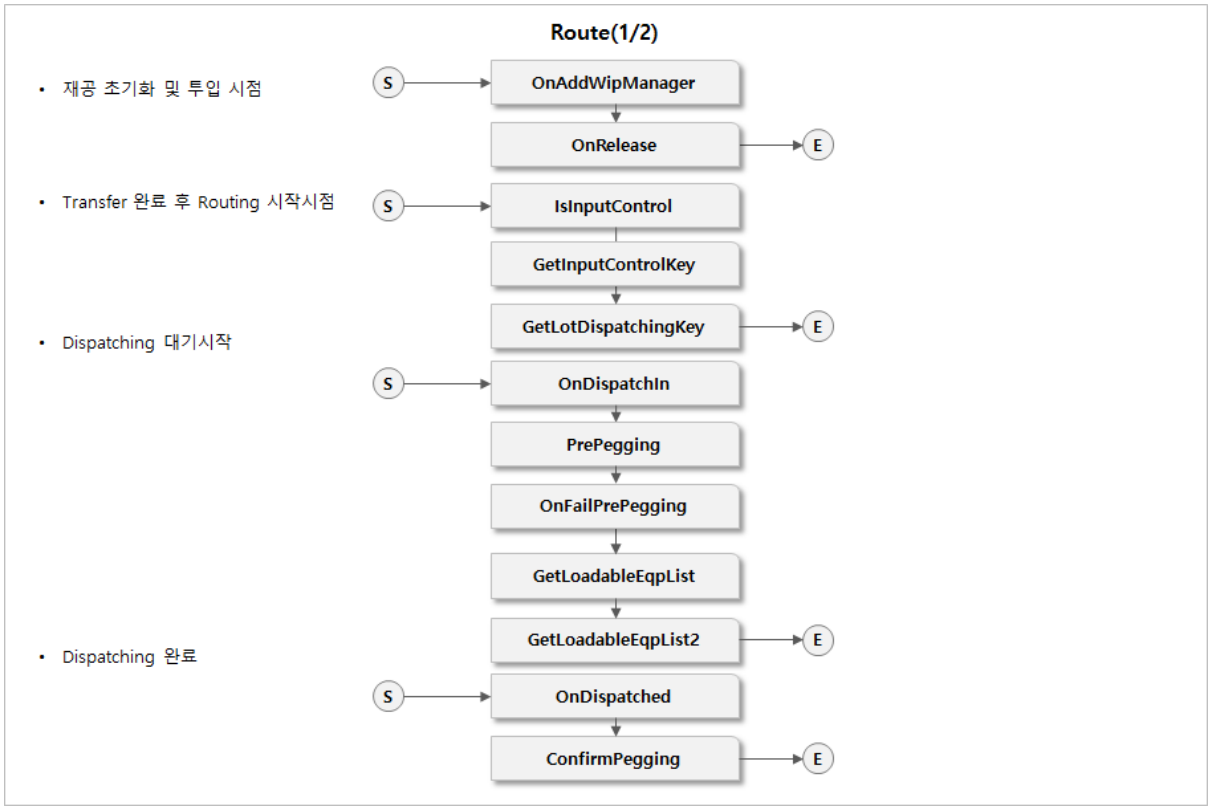
```
public void RESERVE_ONE_DEF(List<ILot> lots, ref int index, ref bool handled)
{
    var batchInitiator = ServiceLocator.Resolve<BatchInitiator>();
    batchInitiator.Reserve(lots[index]);
    index++;
}
```

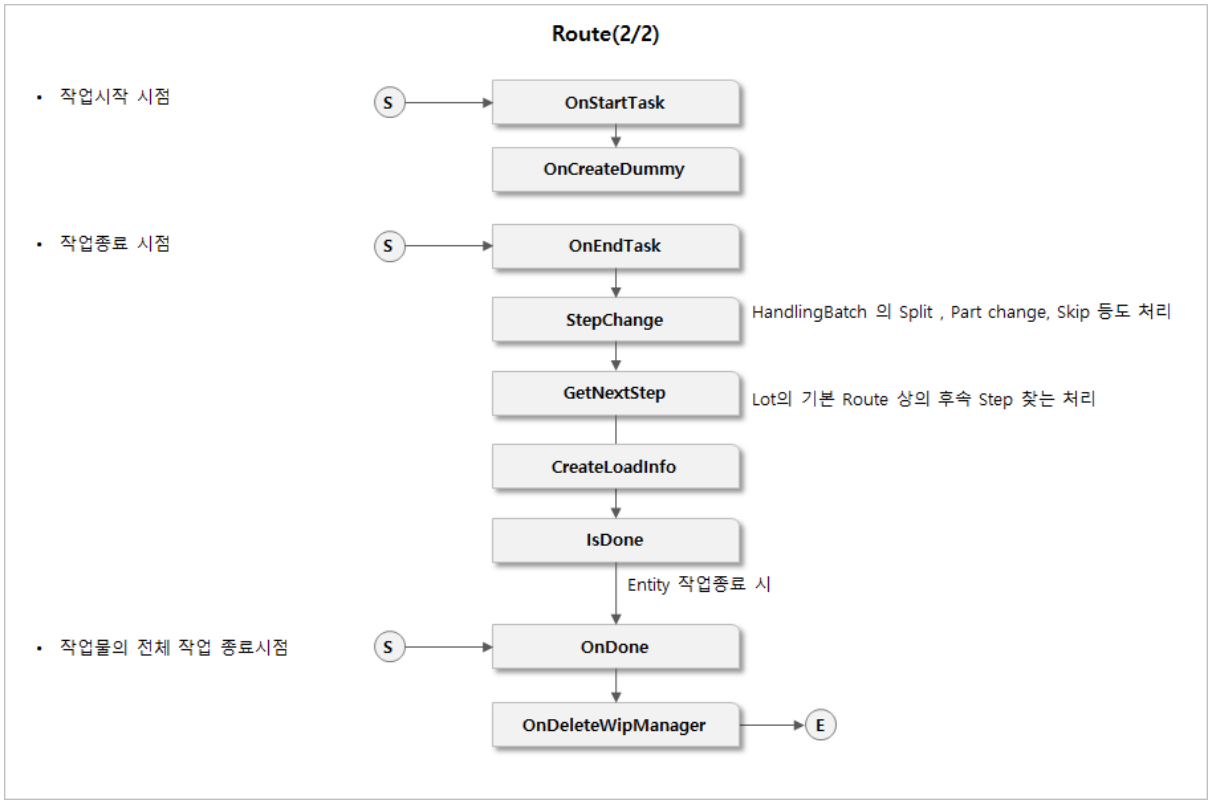
6. ***OnEndReserve*** : 투입대상 작업물을 모두 등록한 후 호출되며, 이 때 필요한 Logic 을 추가 구현합니다.

Route 컨트롤

Route 는 Simulation 수행의 전 주기동안 작업물의 Routing 및 작업상태 결정에 관련한 일련의 Logic 을 담당하는 FEComponent 입니다. 아래 그림은 Route 에서 처리하는 FEAction 의 처리 흐름을 표시한 것입니다. Route 의 경우 연속적인 Logic 의 처리를 수행하는 것은 아님으로 작업물의 속성을 처리하기 위한 주요 이벤트에 대한 Logic 처리를 위한 Action의 집합입니다. 일부 Action 간에 Logic 처리의 연속성이 있는 부분이 있으나 대부분의 Action 을 처리하기 위한 Event 는 단속적으로 호출된다고 할 수 있습니다.

아래는 Route 을 구성하는 Action의 호출 시점별 실행 순서를 도식화한 그림입니다.





작업물은 장비 혹은 Bucket 을 통해 공정을 진행하게 됩니다. 현 공정의 종료 처리를 시작으로 작업이 완료된 작업물은 다음 공정 Routing 을 결정합니다. 이 때 작업물이 여러 개로 분리 (Split) 되거나 다른 작업물과 병합 (Merge) 될 수 있습니다. 다음 공정이 더 없는 경우 작업물

은 소멸되며, 그렇지 않은 경우 물류이동 Rule 에 따라 다음 공정까지 이동한 후 대기하게 됩니다. 이 과정을 처리하는 FEAction 은 다음과 같습니다. 사용자는 작업물의 Routing 처리를 위해 각 FEAction 을 이용하여 사용자 정의 Logic 을 구현할 수 있습니다.

재공 초기화 및 작업물의 신규 투입 시점

1. ***OnAddWipManager*** : WipManager 에 Lot 이 추가되는 시점에 호출되는 함수입니다. 재공 초기화 시점, 신규 작업물의 투입시점에 자동으로 WipManager 에 작업물이 추가되며, Simulation 중간에 작업물이 Split, Merge 되면서 신규로 작업물이 생성되는 경우에 사용자가 WipManager 에 작업물을 추가해주어야 합니다. 추가시점에 호출됩니다.
2. ***OnRelease*** : 작업물이 라인에 투입되는 시점에 호출되어 투입시점의 정보를 작업물 속성에 반영하거나 통계치를 집계하는 사용자 Logic 을 구현할 수 있습니다.

Transfer 완료후 Routing 시작 시점

1. ***IsInputControl*** : 작업물에 현공정에 대해 투입 제어 여부를 판별합니다. true 를 반환하는 경우에 GetInputControlKey 함수로 사용하게 될 InOutAgent 에 대한 Key 값을 요청하게 됩니다.
2. ***GetInputControlKey*** : 작업물에 대해 투입제어가 필요한 경우 투입제어를 담당할 InOutAgent 에 대한 Key 값을 반환합니다. Key 값은 **Factory > FactoryInit > InitializeInOutAgent** 함수를 통해 생성한 InOutAgent 중 대상 Agent 의 Key를 반환합니다. 작업물은 반환된 Key 와 일치하는 등록된 InOutAgent 에 투입됩니다.
3. ***GetLotDispatchingKey*** : 작업물의 Dispatching 을 담당할 Controller 를 선택합니다. 복수 개의 Dispatching Controller 를 사용하지 않는 경우에는 단순히 Bucketing 과 정상 로딩을 구분하는 용도로 사용될 수 있습니다. 다만, 장비 Group 단위의 Dispatching 을 하는 경우에는 작업물의 현재 공정에 따라 Dispatching Controller 를 선택하는 Rule 이 필요하며 이를 본 FEAction을 통해 구현할 수 있습니다.

Dispatching 시작 시점

1. ***OnDispatchIn*** : 작업물의 Dispatching 을 담당할 Controller 를 선택합니다. 복수 개의 Dispatching Controller 를 사용하지 않는 경우에는 단순히 Bucketing 과 정상 로딩을 구분하는 용도로 사용될 수 있습니다. 다만, 장비 Group 단위의 Dispatching 을 하는 경우에는 작업물의 현재 공정에 따라 Dispatching Controller 를 선택하는 Rule 이 필요하며 이를 본 FEAction을 통해 구현할 수 있습니다

2. ***PrePegging*** : 작업물이 Dispatching 되기 전에 Lot 의 Demand 에 대한 Mapping 정보가 필요할 수 있습니다. 제품별 Target 에 따라 해당 재공의 Dispatching 우선순위 혹은 작업여부가 결정될 수 있기 때문입니다. PrePegging 은 실제 **Entity > ForwardPeg** Control 의 수행 완료 후 실행됩니다. 작업물에 설정된 Target 정보에 따라 제품, 공정의 변경을 수행하는 코드를 작성합니다.

   PrePegging 코드 예시

```
// PrePegging 에 대한 구현
public void PREPEGGING_0(IHandlingBatch hb, Step step, object key, ref bool handled)
{
    var slot = hb as SiteLot;
    // Key 는 StepPlan 의 Key 이며 Product 정보라고 가정
    // 작업물의 Product 정보와 Forward Pegging 시 Target 이 되는 StepPlan 의 Key 와 다른 경우 제품정보 변경
    if (slot.Product != key)
    {
        slot.Product = key as SiteProduct;
        slot.Route = slot.Product.Process as SiteProcess;
        // 제품변경과 함께 Step 정보가 함께 변경되는 경우 이를 처리
        if (slot.CurrentStep != step)
        {
            var newPlan = EntityControl.Instance.CreateLoadInfo(slot, step);
            slot.SetCurrentPlan(newPlan);
        }
    }
}
```

3. ***OnFailPrePegging***: **Entity > ForwardPeg** 컨트롤에서 작업물이 Pegging 할 Target 을 못 찾는 경우 PrePegging FEAction이 호출되지 않습니다. 이 경우 PrePegging FEAction 호출이 실패한 사유에 대해 이력을 남기는 사용자 로직 구현을 위한 FEAction 입니다.

4. ***GetLoadableEqpList***: 정상 로딩 대상 작업물의 로딩 가능 장비 List 를 검색하여 반환하는 Logic 을 구현합니다.

5. ***GetLoadableEqpList2***: 정상 로딩 가능 장비가 없을 때 사용할 수 있는 Alternative 로딩 가능 장비 List 를 제공합니다.

> **Note**
> 공정과 장비를 장비 Group 단위로 Mapping 하여 로딩 가능 여부를 제약하도록 관리할 때, 공정의 기본 장비 Group 에 부하가 많은 경우 혹은 해당 장비 Group 의 장비에 문제가 있는 경우에는 동일 공정을 처리할 수 있는 대체 장비 Group 의 장비를 통해 공정을 수행하는 경우 이러한 Alternative 장비 List 기능을 활용할 수 있습니다.

Dispatching 완료 시점

1. ***OnDispatched*** : 작업물이 선택 완료된 시점에 호출됩니다. 보통 선택 완료 후에 선택을 위해 Lot 에 설정된 조건이나 제약에 대한 업데이트가 이루어집니다.
2. ***ConfirmPegging*** : Dispatching 을 위해 F/W 로 Pegging 했던 정보에 대해 선택된 시점에 확정 하는 작업을 수행합니다. Library 에서 Confirm 작업을 수행하며 사용자가 추가적으로 처리할 Logic 에 대해 본 함수를 구현합니다. 기본적인 처리 내용은 1) 작업물의 CurrentPlan 에 설정된 PegInfoList 의 Target 에서 Pegging 된 수량을 차감하고 2) 작업물의 CurrentPlan 에 설정된 PegInfoList 를 삭제합니다.

현공정의 작업 시작 시점

1. ***OnStartTask*** : 작업물이 작업을 시작할 때 공정시작 처리 Logic 을 구현합니다. 실제 작업 시작은 장비에서 수행하지만, 작업물에 작업시작 시 처리해야 할 정보를 업데이트해야 합니다. 본 Action 은 장비 혹은 Bucketing 시 호출이 가능하며, 처리하는 방식은 같습니다. SEEPLAN Library 에서 기본적인 작업 시작처리를 수행하나, Project 에서 작업물에 추가적인 속성을 처리하기 위해서 본 Action 을 구현할 수 있습니다.
2. ***OnCreateDummy*** : 장비에서 작업물이 작업중에 PM, Down 이 발생하는 경우 Down 직후에 작업중이던 작업물을 연속해서 작업해야 하며, 이경우에 재로딩 되는 Lot 의 Loading 이력이 최초 투입된 Lot 의 이력과 중복되는 것을 막기 위해 Dummy Lot 을 만들어 재투입합니다. 이때 재 투입되는 Lot 이 만들어지는 시점에 호출됩니다.

현공정의 작업 완료 시점

1. ***OnEndTask*** : 작업물이 작업을 종료할 때 공정완료 처리 Logic 을 구현합니다.
2. ***StepChange*** : 작업물이 현 공정을 마치고 다음공정을 찾을 때의 Logic 을 구현합니다. ***MoveNext*** 시점에 작업물의 Part change 에 따른 Split 과 같이 분리되어 작업될 수 있는 사용자 Rule 을 구현할 수 있습니다. MoveNext 는 물리적으로 Entity 가 물리적으로 이동하기 전에 호출되어 작업물이 다음에 진행해야 할 공정을 설정합니다. 이후 ***IsDone*** 을 호출하게 됩니다.
3. ***GetNextStep*** : 작업물의 공정 흐름상의 다음작업을 찾아 반환합니다. 일반적인 제품별 Process 가 정의된 경우 다음 작업을 호출하게 되며, 특정 공정에 대한 Skip 및 별도의 Logic 에 의해 Next Task 를 찾아야 하는 경우 사용자 정의 Logic 을 구현합니다.

   Default Logic Example

```
 public Step GET_NEXT_STEP_DEF(ILot lot, LoadInfo cLoadInfo, Step cStep, DateTime no
w, ref bool handled, Step prevReturnValue)
 {
     Step next = cStep.GetDefaultNextStep();
     return next;
 }
```

4. ***CreateLoadInfo*** : 작업물의 신규 설정된 공정에 대한 작업정보를 생성하는 코드를 작성합니다. 작업물의 속성인 LoadInfo 의 경우 작업물이 시뮬레이션 이후 거쳐간 모든 공정에 대해 하나씩 생성되며, 작업당시의 정보를 저장하고 있습니다. 기본적으로는 작업을 수행한 Resource, 작업시작시각, 종료시각 등의 정보를 포함하나 사용자가 LoadInfo를 **My Objects** 의 기능을 통해 상속받아 새로 정의한 경우 사용자 속성을 추가할 수 있습니다. 본 함수는 시뮬레이션 라이브러리내에서 Step 변경시 자동으로 호출이 됩니다. 그러나, 사용자가 로직처리 임의 시점에 현재 Lot의 CurrentPlan 을 변경할 필요가 있는 경우에는 Lot의 **SetCurrentPlan** 함수를 사용할 수 있습니다.

5. ***IsDone*** : MoveNext 이후에 호출되거나, 재공 초기화 시점에 호출됩니다. 재공 초기화시점에는 재공의 초기 상태가 종료대상인 경우가 존재함으로 이를 처리하기 위해 호출됩니다. Step route 상의 모든 공정을 종료하여 소멸대상이 되는지 여부를 판별하는 로직을 구현합니다. 일반적으로 다음 공정을 찾지 못하는 경우 종료한 것으로 판별하나 구현대상에 따라 작업종료 공정이 지정되는 경우도 있어 이를 고려한 로직을 구현할 수 있습니다.

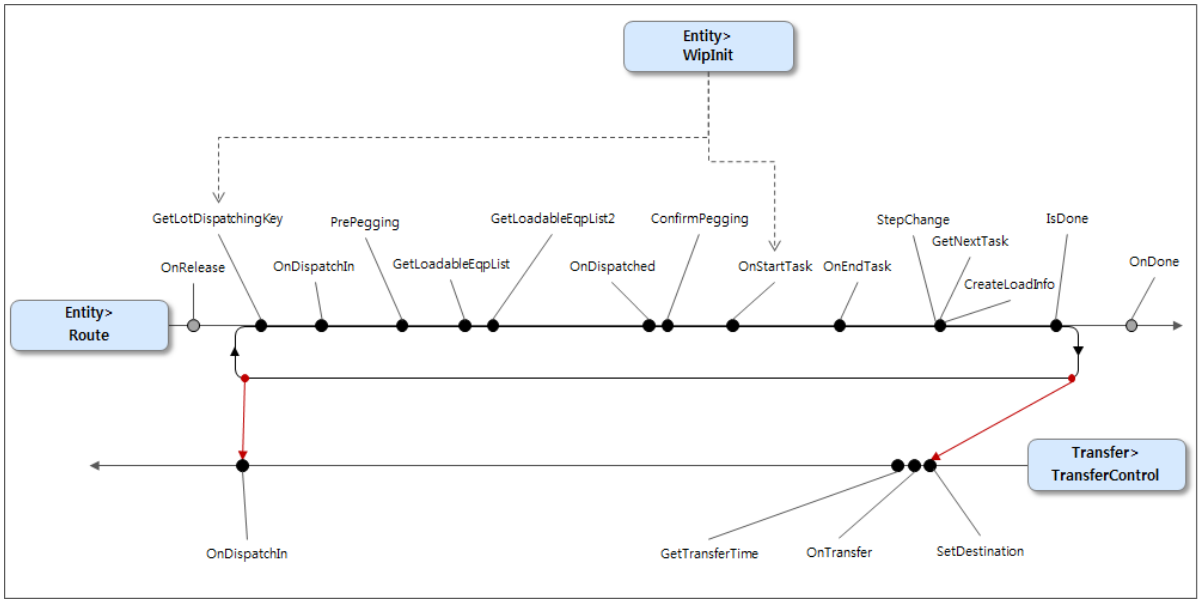
   Default Logic Example

```
public bool IS_DONE_DEF(IHandlingBatch hb, ref bool handled, bool prevReturnValue)
{
   return hb.IsFinished;
}
```

6. ***OnDone*** : 작업물이 라인에서 작업을 모두 마치고 소멸되는 시점에 호출되며, 통계수집 등의 사용자 정의 Logic 을 구현할 수 있습니다.

7. ***OnDeleteWipManager*** : Lot 의 모든 작업이 끝나고 소멸되는 시점에 WipManager 에서 제거한후 호출되는 함수입니다. Logic 검증 등의 목적으로 사용할 수 있습니다.

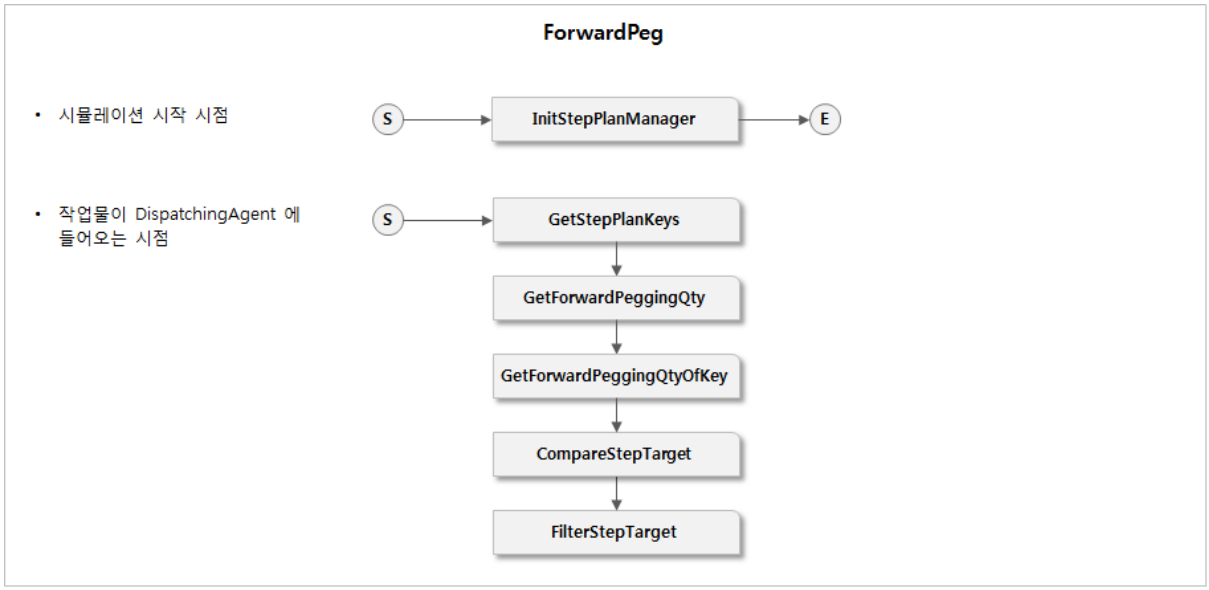
연관된 Control 들의 Action 들과 연관관계는 다음과 같습니다.



ForwardPeg 컨트롤

ForwardPeg 는 작업물이 여러 제품으로 생산이 가능한 경우 제품 분기되는 지점에서 등록된 제품별 생산목표 (Target) 중 어떤 제품의 목표를 위해 생산할 것인지를 결정하는 과정을 컨트롤하기 위한 FEComponent 입니다. 제품이 제품 분기 공정에 도착한 후 로딩 가능한 장비를 찾아 대기하기 전에 수행되는 Logic 입니다.

아래는 ForwardPeg 에 포함된 FEAction 의 호출 시점별 실행 순서를 도식화한 그림입니다.



Simulation 시작 시점

1. ***InitStepPlanManager*** : Forward Pegging 을 위해 Backward Pegging 시 산출된 Step 별 Target 을 관리하는 클래스가 StepPlanManager 입니다. 본 Action 에서 StepPlanManager 를 초기화합니다. Pegging Module 의 WriteTarget Rule 에서 추가된 StepPlan 과 StepTarget 을 가공하거나 새로운 StepPlan 과 StepTarget 을 추가할 수 있습니다.

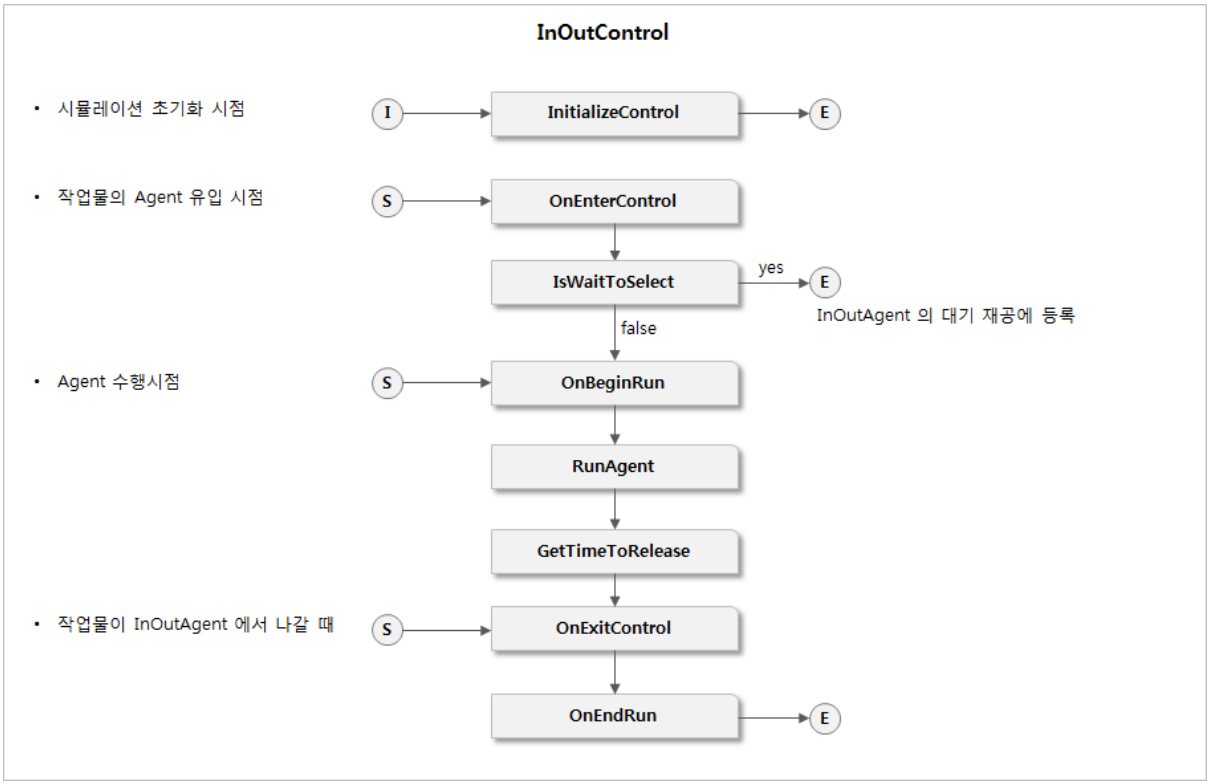
Forward Pegging 시점 (작업물이 DispatchingAgent 에 들어오는 시점)

1. ***GetStepPlanKeys*** : 작업물에 대한 StepPlan 을 찾기 위한 Key 의 목록을 반환합니다. 첫 번째 Key 를 갖는 StepPlan 이 존재하면 Forward Pegging Logic 에서는 해당 StepPlan을 사용하고, 존재하지 않으면 다음 Key 의 StepPlan을 사용합니다. 모든 Key 의 StepPlan 이 존재하지 않으면 ForwardPegging Logic 이 실행되지 않습니다.
2. ***GetForwardPeggingQty*** : 작업물의 특정 Target 에 대해 Forward Pegging 가능한 수량을 반환합니다. 일반적으로 Target 이 큰 경우 전량 Pegging 할 수 있으나 경우에 따라 작업물의 일부만 Pegging 가능함으로 Pegging 수량을 설정합니다.
3. ***GetForwardPeggingQtyOfKey*** : GetStepPlanKeys의 목록 별 Pegging 가능한 수량을 반환합니다. StepPlanKey별로 Lot 수량의 전량을 Pegging 할 수 있는 경우도 있지만 Key 별로 가능수량을 수량을 따로 제어할 수도 있습니다. Pegging 가능한 총량은 GetForwardPeggingQty 수량을 넘지 못합니다.
4. ***CompareStepTarget*** : 작업물이 Pegging 가능한 StepTarget 이 여러 개인 경우에 우선순위를 정의합니다. 본 FEAction은 StepPlanManager.SortStepTargets = true일때만 호출됩니다. StepPlanManager.SortStepTargets의 기본값은 true입니다.
5. ***FilterStepTarget*** : 작업물이 StepTarget 에 Pegging 될 수 없는 조건인지를 체크합니다. 작업물이 Pegging 가능한 모든 Target 별로 평가합니다. true 인 경우 Filter 가 적용되어 제외됩니다.

> ⚠️ **Warning**
> Forward Pegging 은 상기의 5개 Action 이 처리될 때 수행되어 작업물의 Target 이 결정됩니다. 이 경우에 만일 Pegging 된 제품 (Product) 과 현재 작업물의 제품이 다른 경우 ***Entity > Route > PrePegging*** 함수에서 제품을 변경하고 Log 를 남기는 등의 작업을 수행합니다.

InOutControl 컨트롤

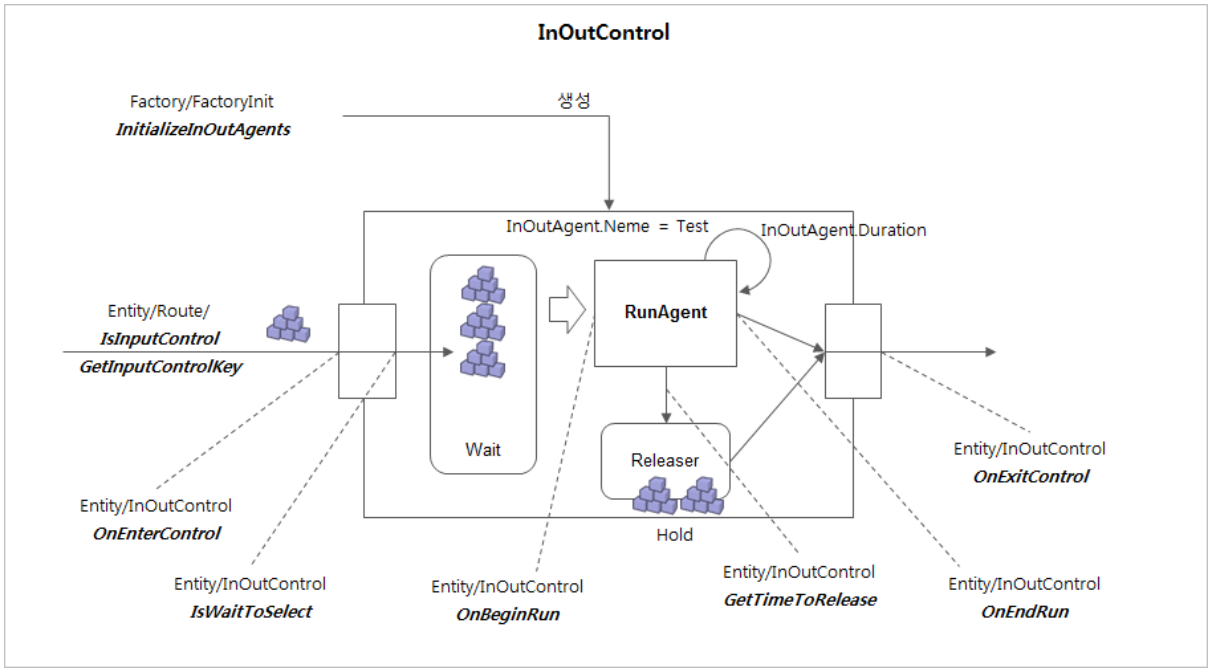
InOutControl 은 특정 구간이나 라인의 투입 지점에서 투입목표 (Target) 및 투입제약에 따라 투입 양을 결정하고 투입대상 작업물을 선정하여 투입하도록 제어하기 위한 FEComponent 입니다. 작업물의 최초 투입시점에는 통상 투입 Target 에 기반하여 투입 작업물을 생성하는 투입하도록 처리합니다. 그러나 재공 중 특정 Step 에서 이후 공정 구간에 진입여부를 판별하기 위한 경우에는 Entity > Route > IsInputControl, GetInputControlKey 함수를 통해 재공을 모아두었다가 주기적으로 구간의 Capacity 및 제약을 고려하여 투입여부를 결정하는 방식으로 사용합니다.



1. ***InitializeControl*** : InOutAgent 에 대한 초기화 Logic 을 처리합니다.
2. ***OnEnterControl*** : 작업물이 투입제어를 위해 Agent 에 유입되는 시점에 호출됩니다. 작업 Log 를 기록하거나 유입되는 작업물의 속성들을 편집, 통계수집 등의 용도로 사용됩니다.
3. ***IsWaitToSelect*** : 작업물이 투입제어를 위한 Agent 에 유입되는 경우 Agent 의 실행 주기에 해당하지 않을 때 대기여부를 판별합니다.

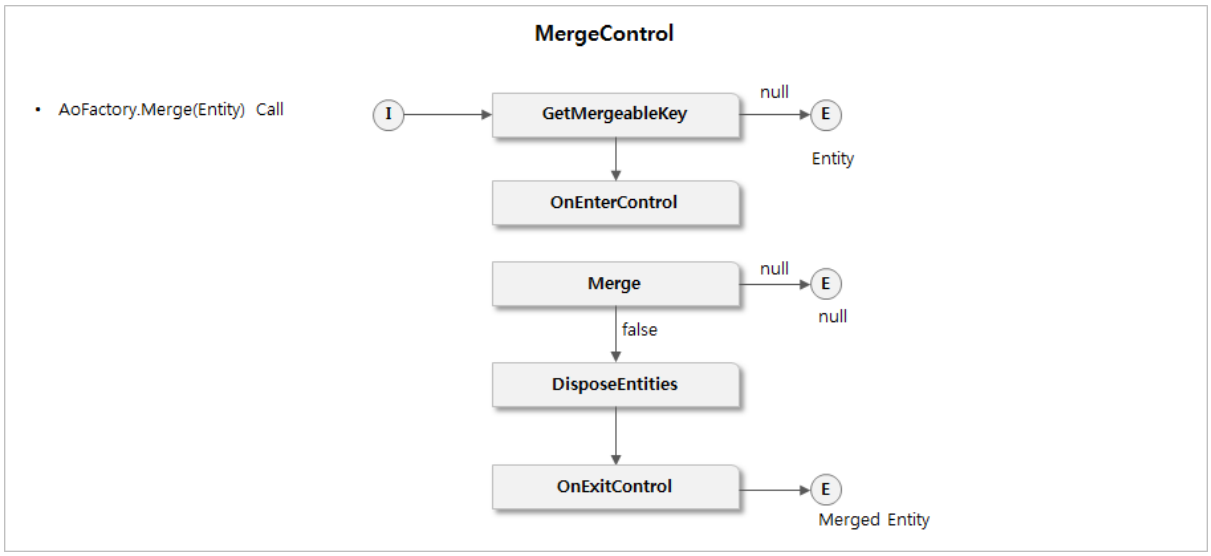
4. ***OnBeginRun*** : Agent에서 투입대상 작업물 (생성) 선택 Logic 실행 시작 시점에 호출됩니다.
5. ***RunAgent*** : Agent 의 Logic 을 구현합니다. 아래는 예시적인 구현 시나리오입니다.
   - *최초 작업물의 투입시점 처리* : 제품별 투입 Target 으로부터 투입 가능 수량만큼 작업물을 생성한 후 투입하도록 처리합니다. 경우에 따라 투입 대상 작업물이 이미 생성되어 있는 경우에는 생성부분을 제외한 Logic 을 구현합니다.
   - *구간 투입처리 시점* : 수행 주기 사이에 Agent 로 유입되어 투입 대기 중인 작업물의 우선순위 및 제약 위배 여부를 판별하여 투입 가능 작업물을 선별합니다.
6. ***GetTimeToRelease*** : RunAgent 가 수행되는 시점에 반환된 작업물을 실행시점에 Agent 를 빠져나갑니다. 만일 일정 시간이후에 Agent 를 나가도록 처리하려는 경우에는 InOutAgent.AddToRelease() 함수를 사용하여 일정시점 이후에 Agent 를 빠져나가도록 처리할 수 있습니다. AddToRelease 함수로 작업물을 넘겨줄때 GetTimeToRelease 함수가 호출되며, 이함수에 실제 Release 될 시간을 반환하도록 로직을 구현합니다. 반환값은 대기시간 입니다.
7. **OnExitControl** : RunAgent Logic 에서 반환된 작업물을 Agent 에서 내보내는 시점에 호출됩니다.
8. **OnEndRun** : Agent Logic 의 주기별 실행 직후에 호출됩니다.

아래 그림은 InOutAgent 의 동작 원리를 도식화한 그림입니다. Factory 초기화시점에 Agent 의 명칭과 수행주기를 결정하고, 작업물의 이동시점에 Agent 로 들어온 경우 Agent 의 주기에 따라 Lot 의 투입을 결정하여 내보내는 구조이며, 내부적으로 라인 투입 의사결정 시점에 일정 시간 이후에 라인에 투입하려 할 경우 내부 Releaser 에 작업물을 등록하고 일정 시간 경과후 라인에 투입하도록 모델링 할 수 있습니다.



MergeControl 컨트롤

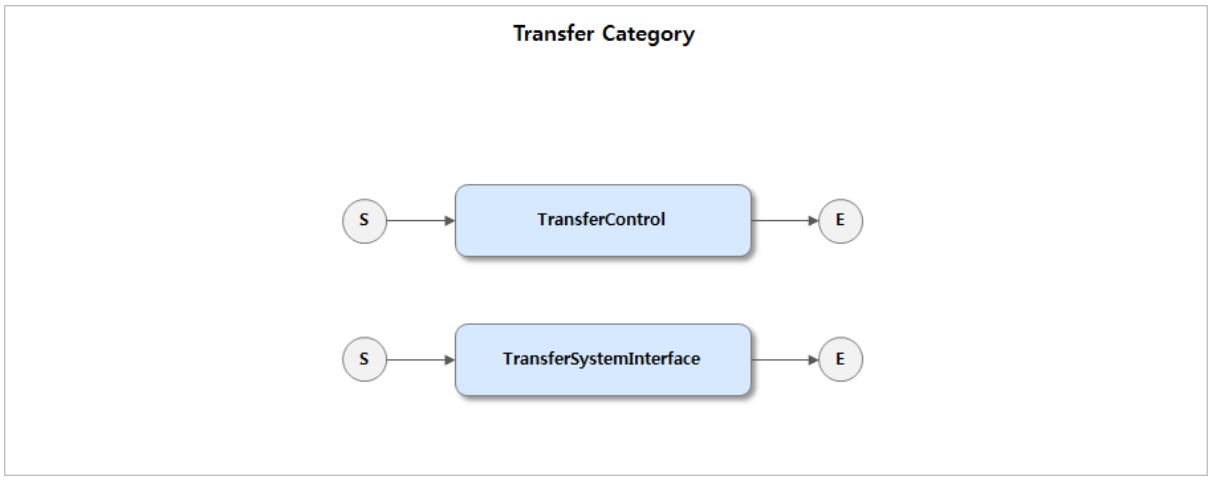
MergeControl은 작업물이 공정 특성이나 기타 사유에 의해 여러개의 작업물로 분할되어 있는 작업물을 하나로 Merge 시키는 역할을 수행하며, 이와 관련된 일련의 Logic 을 사용자가 개발할 수 있도록 지원하는 Component 입니다. 작업물의 Merge는 여러 FEAction에서 발생할 수 있기때문에 사용자가 AoFactory.Merge() 함수를 호출할 때 사용되도록 구성되어 있습니다. 그리고 Merge 함수를 호출하면 MergeControl을 통해 만들어진 작업물을 반환받을 수 있으므로 이후 작업물의 진행 여부는 사용자가 직접 결정할 수 있습니다.



1. **GetMergeableKey** : Merge해야할 작업물을 Key 별로 모아두는 데 이때, 작업물을 분류하는 역할로 사용됩니다. Key 가 Null로 설정되는 경우, 작업물이 Merge 대상으로 등록되지 않고, 바로 해당 작업물을 넘게 주게 됩니다.
2. **OnEnterControl** : Merge 대상으로 들어온 작업물에 설정이 필요하거나 결과 작성을 위한 FEAction 입니다. GetMergeableKey가 Null이 아닐 경우 호출이 되며, MergeControl에 작업물이 등록되게 됩니다.
3. **Merge** : Merge 대상들을 Merge 하는 역할을 수행합니다. 이 때 작업물이 Merge 가능 여부를 체크하고 가능하다면, Merge 를 해서 반환해야 합니다. Merge가 된 작업물이 없다면 Null, 작업물이 Merge가 되었다면 Merge 한 작업물을 반환하도록 구현되어야 합니다.
4. **DisposeEntities** : Merge 된 작업물을 삭제하는 역할을 수행합니다. Merge가 완료되어 Merge 가능리스트에서 삭제가 필요하다면, 삭제해야 할 대상을 찾아 반환합니다. 삭제할 대상이 없을 경우 Null 을 반환하면 됩니다. Merge가 완료된 작업물을 삭제하지 않을 경우, Merge FEAction 호출 시 Merge 가능한 리스트로 사용되기때문에 별다른 처리가 없을 경우 작업물이 중복되어 사용될 수 있습니다.
5. **OnExitControl** : Merge 가 완료된 후 수행하는 FEAction 입니다.

Transfer

작업물의 From-To Location 간 이송을 제어합니다. 작업물의 이송과 관련된 결과분석 요구사항에 따라 물류에 대한 단순한 Model 을 적용하거나 외부의 물류 Simulation 도구와의 I/F 를 통해 정교한 물류 제어에 기반한 Model 을 적용할 수 있습니다. 다음 그림은 Transfer Category 의 Component 구성을 나타냅니다.



TransferControl

작업물이 작업을 마치고 다음 공정이 설정된 상태에서 이송에 대한 물류처리를 담당합니다. 물류 이송에 대한 통계수치 (평균 이송 시간, 평균 대기 작업물 수량 등) 의 집계가 목적인 경우에는 이송 장치에 대한 좀 더 세부적인 Model 을 Logic 상 추가할 수 있으며, 그 외의 경우 단순한 이송 시간 소요를 위한 Delay 처리에 대한 Logic 을 반영할 수도 있습니다.

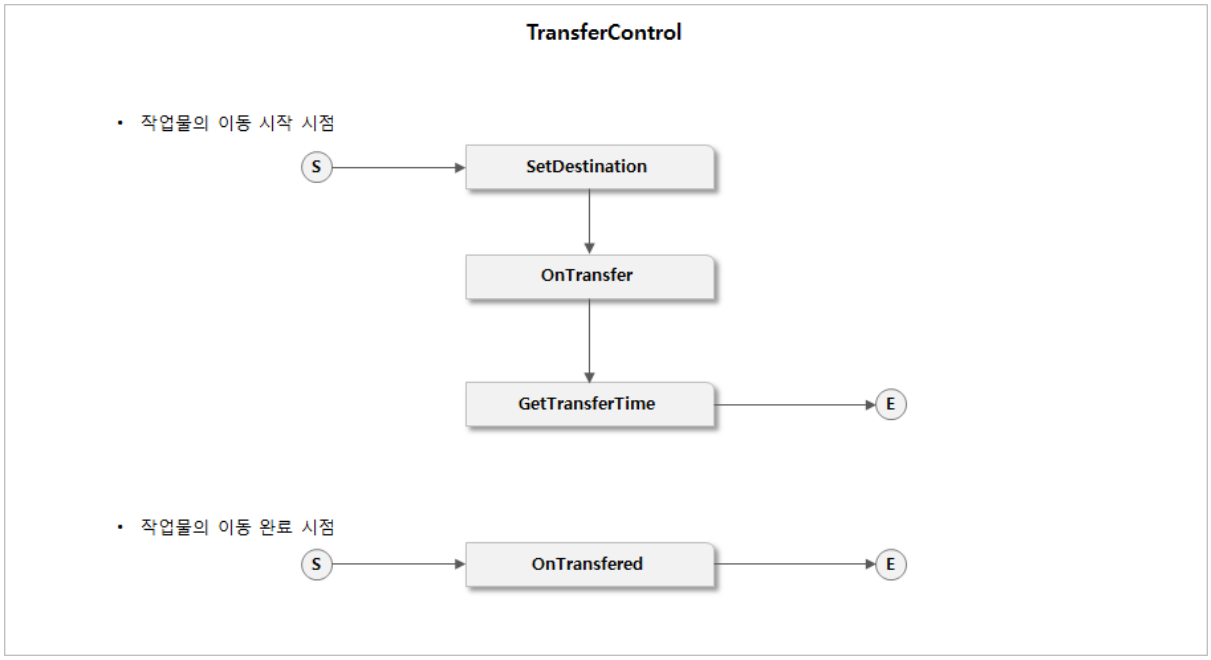
TansferSystemInterface

물류 이송 장치에 보다 상세한 Simulation 을 수행하는 경우 물류 Simulation 시스템과 연동하여 생산 Simulation 을 수행할 수 있도록 Interface Logic 을 구현할 수 있습니다.

TransferControl 컨트롤

Transfer Control 은 작업물의 이송 시작 시점부터 이송, 이송 완료 시점까지 작업물을 제어하기위한 FEComponent 입니다. 자동화된 생산 라인의 경우 작업물은 AMHS (Automated Material Handling System) 을 통해 이송이 이루어지며 Transfer Control 는 이러한 AMHS 의 역할을 수행합니다. 물론 자동화 수준이 낮아 작업자에 의해 물류가 처리되는 경우에도 동일한 방식으로 모델링이 가능합니다. 의사결정 및 실행의 주체가 달라지는 차이가 있을 뿐입니다. TransferControl 은 목적에 따라 물류시간만을 고려한 비교적 단순한 물류처리 방식을 지원하도록 구성됩니다.

작업물은 이전 공정에서의 작업을 마친 후 Step Route 에 의해 다음공정을 결정합니다. Simulation 의 Transfer 처리 Model 에서 공정과 장비 정보를 기반으로 다음공정의 이송 위치 정보를 찾고, 이전 위치에서 다음 위치까지의 이송 시간을 산출합니다. 산출된 이송 시간이 경과한 후 작업물은 이송 완료되며, 이 때 작업물의 최종 위치 정보 및 변경이 필요한 속성 정보를 업데이트하거나 통계를 수집하는 등의 작업을 수행합니다.



작업물의 공정간 이동 시작부터 이동 종료까지 외부 물류시스템과의 연계가 없는 상황에서 시간기준의 물류 제어를 수행합니다.

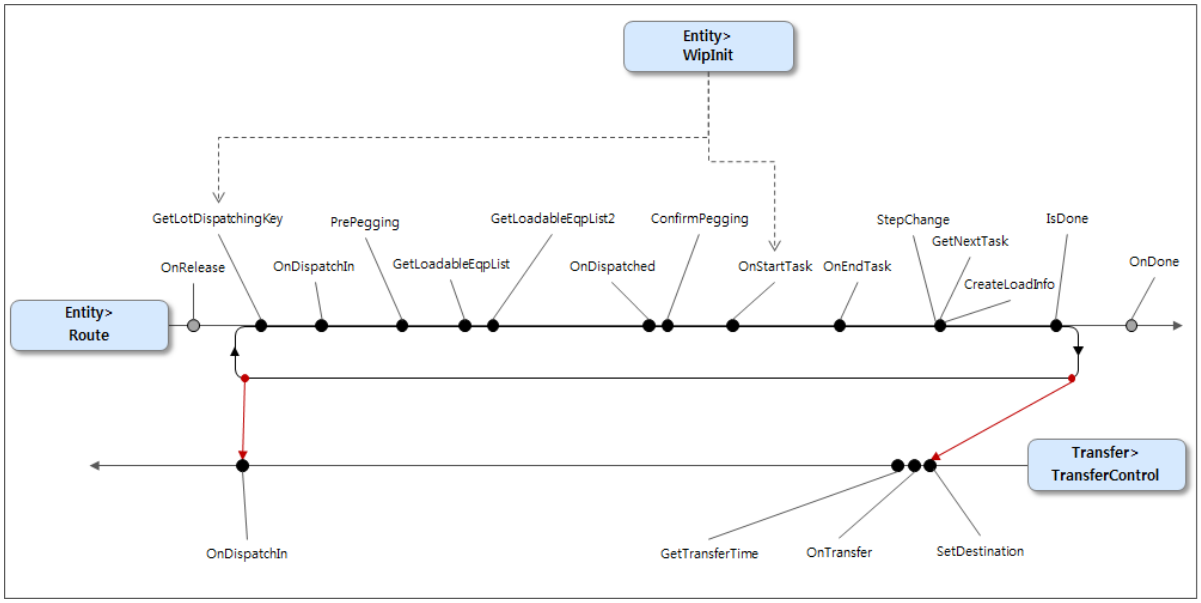
1. ***SetDestination*** : 작업물이 이송을 시작하기 직전에 공정정보와 로딩가능 장비정보를 기준으로 이송 목적지를 결정하기 Logic 을 구현할 수 있습니다.
2. ***OnTransfer*** : 작업물의 이송을 시작시점에 호출되는 FEAction 으로 이송 관련 통계치의 수집 및 작업물의 속성변경을 수행합니다.
3. ***GetTransferTime*** : 이송 시간을 산출하여 반환하는 함수입니다. 사용자 정의에 의해 이송 시작 지점, 목표 지점간의 이송 시간을 산출하는 Logic 을 구현합니다. Model 에서는 Simulation Model 이 Transfer 시작 시점에 작업물을 Hold 하고 있다가 산출된 이송 시간이 경과한 후 작업물을 Release 하도록 구성되어 있습니다.

   Default Logic Example

   ```
   public Time GET_TRANSFER_TIME_DEF(Mozart.SeePlan.Simulation.IHandlingBatch hb, ref b
   ool handled, Time prevReturnValue)
   {
      SeeplanConfiguration conf = ServiceLocator.Resolve<SeeplanConfiguration>();
      return Time.FromMinutes(conf.TransferTimeMinutes);
   }
   ```

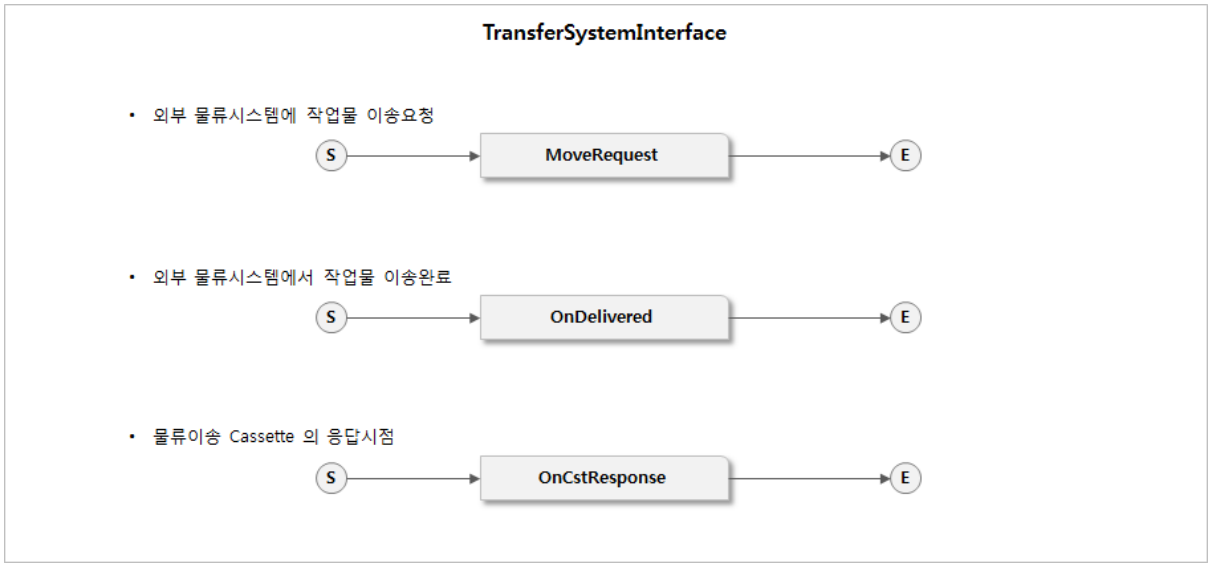
4. ***OnTransfered*** : 작업물 이송이 완료되는 시점에 호출되는 FEAction 입니다. 이송 관련 통계치의 수집 및 작업물의 속성변경을 위한 사용자 정의 Logic 을 구현할 수 있습니다.

연관된 Control 들의 Action 들과 연관관계는 다음과 같습니다.



TransferSystemInterface 컨트롤

이송 장치에 대한 Capacity 검토 및 물류에 대한 이슈가 많은 경우 Simulation 에서 이러한 목표수치를 산출할 수 있도록 물류 이송에 대한 정교한 모델링이 필요합니다. Crane, AGV, Conveyor 등의 자동화된 물류이송 장치와 이들을 컨트롤할 수 있는 Logic 을 포함하는 Transportation Simulation 을 제공하는 다양한 시스템이 존재하며, MOZART Engine 은 이러한 물류 Simulation 시스템과의 연동을 통해 보다 현실적인 물류이송 Model 을 반영한 Simulation 을 수행할 수 있도록 연동기능을 제공합니다.



외부 물류 Simulation 시스템과의 연동 시에는 외부 시스템과의 Interface 를 구현한 외부 시스템을 등록한 후 Transfer Control 의 다음 두 가지 FEAction 을 통해 연동을 위한 Logic 을 구현할 수 있습니다. TransportAdapter 는 System 등록 시 자동으로 생성됩니다. ITransportSystem 의 등록 방법은 Manual 을 참조 바랍니다.

1. ***MoveRequest*** : External Transport System 에게 이송 요청을 하도록 구현해야 하며, 이 때 TransportAdapter 객체의 MoveRequest 를 활용합니다.
2. ***OnDelivered*** : 외부 시스템에 의해 Simulation 된 결과로 이송이 완료되는 시점에 호출되는 FEAction 으로 외부 시스템에 의한 이송 중인 동안 TransportAdapter에서 대기 중이던 작업물을 찾아 이송 종료하도록 처리합니다.

   Default Logic Example

   ```
   public void ON_DELIVERED_DEF(TransportAdapter ta, string key, string sourceLocation,
   string targetLocation, ref bool handled)
   {
      // 작업물을 Key 로 찾아야 하는데 여기에 구현이 필요한지 잘 모르겠음
      ISimEntity entity = ServiceLocator.Resolve<ISimEntity>();
      if (ta.Fifo.Contains(entity))
      {
          ta.Factory.Transfer.TakeOut(entity);
      }
   }
   ```

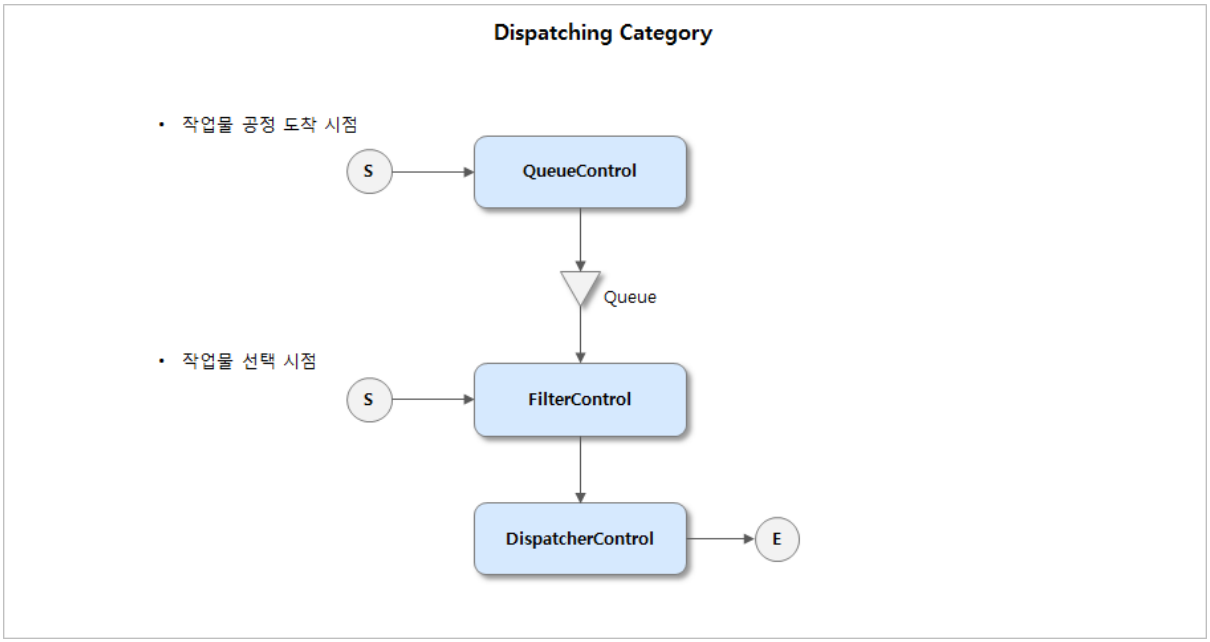
3. ***OnCstResponse*** : 작업물 Container (Cassette) 에 대한 이송호출 응답 시점에 실행되는 Logic 을 구현합니다.

Dispatching

Dispatching Control Category 는 1) 작업물이 공정에 도착하는 시점에 작업물 제어를 위한 QueueControl, 2) 작업물의 Dispatching 시작 시점에 장비와 작업물의 조건을 확인하여 Dispatching 대상만을 Filtering 하는 FilterControl, 3) 최종 Dispatching 대상 작업물의 우선순위를 평가하고 선택하는 DispatcherControl 의 세 부분으로 구성됩니다. Dispatching 시점의 대기 재공만을 평가하는 경우 유입되는 재공이나 후속 공정의 Capacity/Load 상황을 고려한 스케줄을 하기 어려워집니다. 따라서 전,후 공정의 Capacity Balance 를 고려한 Job Change 시점의 결정을 위해 **JobChangeAgent** 와 함께 Logic 을 구성하는 경우도 있습니다.

Dispatching Control 의 시작 Event 는 두 가지로 작업물이 공정에 도착하는 Event 와 장비가 작업물을 선택 (Dispatching) 하는 Event 입니다. 작업물이 공정에 도착해서 해당 작업물의 로딩 가능 장비 중 작업물을 로딩할 수 있는 상태의 장비가 있는 경우 작업물을 선택하는 Event 가 연속해서 발생합니다. 그러나 로딩 가능 장비가 모두 가동중이거나 Down 상태로 가용한 장비가 없는 경우 작업물은 대기하게 되며, 장비가 Idle 상태가 되어 작업물을 로딩할 수 있는 시점에 Dispatching Event 가 발생하고 DispatchFilterControl, DispatchControl 순으로 Logic 을 처리한 후 제어를 종료합니다.

아래 그림은 Dispatching Category 의 Component 구성을 도식화하여 설명합니다.



QueueControl

작업물이 이전 공정에 도착하는 시점에 작업물 제어를 위한 FEComponent 입니다. 최종적으로 작업물을 장비 Queue 에 할당하면 제어가 끝니다.

FilterControl

Dispatching 시점에 장비에서 Dispatching 을 할 수 없는 상태이거나 작업물 중 제약조건에 의해 작업이 불가한 작업물을 제외시키도록 제어하는 FEComponent 입니다.

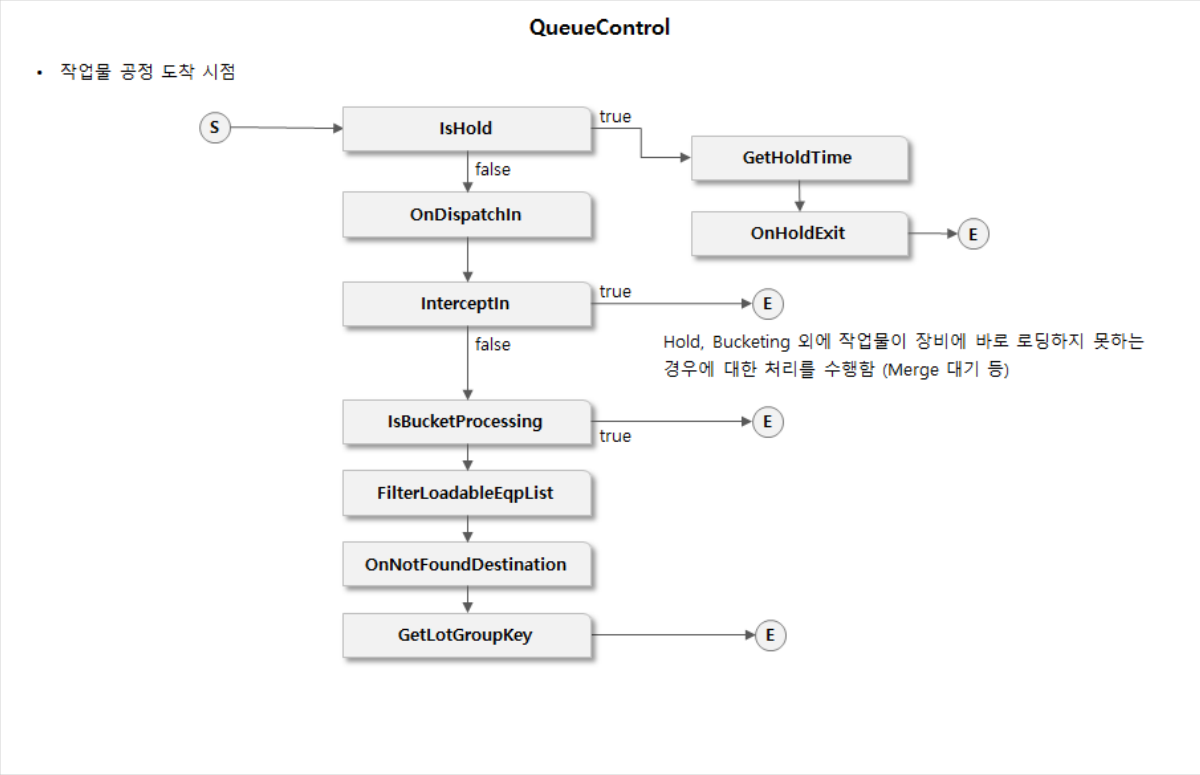
DispatcherControl

장비가 작업물을 선택해야 하는 시점에 대기 중인 작업물의 우선순위를 평가하고 장비의 Processing 방식과 평가된 우선순위에 따라 작업물을 선택하기 위한 FEComponent 입니다.

QueueControl 컨트롤

작업물이 공정에 도착하면 일반적으로 해당 공정의 로딩 가능한 장비를 찾아 대기하다가 Dispatching 시점에 평가를 통해 선택되거나 장비가 사용 가능해질 때까지 대기하게 됩니다. 그러나 작업물의 조건에 따라 해당 공정에서 일정시간 동안 Hold 한다거나, 작업물이 복수의 작업물로 분리 혹은 병합되어 작업물이 재구성될 수도 있습니다. 또한, 만일 해당 작업물이 특정 장비에 작업예약되어 있는 경우라면 정상적인 처리 순서를 건너뛰고 바로 장비에 투입될 수 있도록 처리되어야 합니다.

QueueControl 은 작업물이 공정에 도착하는 시점에 처리되는 일련의 작업을 컨트롤하기 위한 FEComponent 로 다음 FEAction 을 순차적으로 호출하며, 사용자는 구축 사이트의 운영방식에 맞도록 해당 FEAction 을 재정의 할 수 있습니다.



1. ***OnDispatchIn*** : 작업물이 QueueControl 의 이후 Logic 을 수행하기 전에 호출되며, 이 시점에 처리가 필요한 작업물이나 기타 정보의 상태를 변경하는 코드를 작성합니다.
2. ***IsHold*** : 작업물이 현 공정에서 Hold 처리되어야 하는지 여부를 판별합니다.
3. ***GetHoldTime*** : IsHold 결과 Hold 처리가 되어야 하는 경우 Hold 시간이 정해져야 하며, 본 FEAction 을 통해 대상 작업물의 속성을 확인하고 Hold 시간을 산정하여 반환하도록

구현합니다. Hold 시간 이후 작업물은 QueueControl 에 의해 처음부터 제어를 다시 시작하게 됩니다.

4. ***OnHoldExit*** : Hold 시간 이후 Dispatching Queue 로 작업물이 들어갈 때 실행되는 Action 입니다.
5. ***InterceptIn*** : 작업물이 공정에 도착한 후 정상적으로 대기 하기 전 Merge 혹은 예외 조건에 해당하는지 점검하고 필요 시 예외 처리하는 Logic 을 구현합니다. 본 Action 의 결과가 true 인 경우, 작업물은 이후 Logic 을 진행하지 않고 별도로 처리되어야 합니다.
6. ***IsBucketProcessing*** : 작업물의 현 공정이 Bucketing 으로 처리되어야 하는지 판단하는 코드를 작성합니다. Bucketing 으로 처리하는 경우에는 이후 Logic 을 진행하지 않습니다. Bucketing 으로 처리하는 경우 Library 상에서 Bucketer 로 작업물을 진행하도록 처리함으로 사용자가 별도의 처리를 추가할 필요는 없습니다.
7. ***FilterLoadableEqpList*** : 로딩 가능 장비 List 에 대한 제약이 존재할 경우 제약을 반영하여 로딩 가능 장비 List 중 제약장비를 제외하는 Logic 을 구현합니다.
8. ***OnNotFoundDestination*** : 장비에서 처리해야 하는 작업물인데 로딩 가능 장비가 없는 경우, 해당 작업물을 처리하는 Logic 을 정의합니다. 일반적으로 이런 경우 기준정보가 잘못 설정되었거나 업데이트가 안되서 발생하는 문제일 확률이 높습니다. 따라서 해당 공정의 TAT 를 기준으로 Bucketing 처리하여 작업을 진행하도록 Rule 을 정의할 수 있습니다. 그러나 Simulation 이 끝날 때까지 대기하도록 처리하여 기준정보 및 Simulation 의 문제점을 분석하는 용도로 사용할 수도 있습니다. 아래는 Bucketing 처리 예시입니다.

   구현 예시

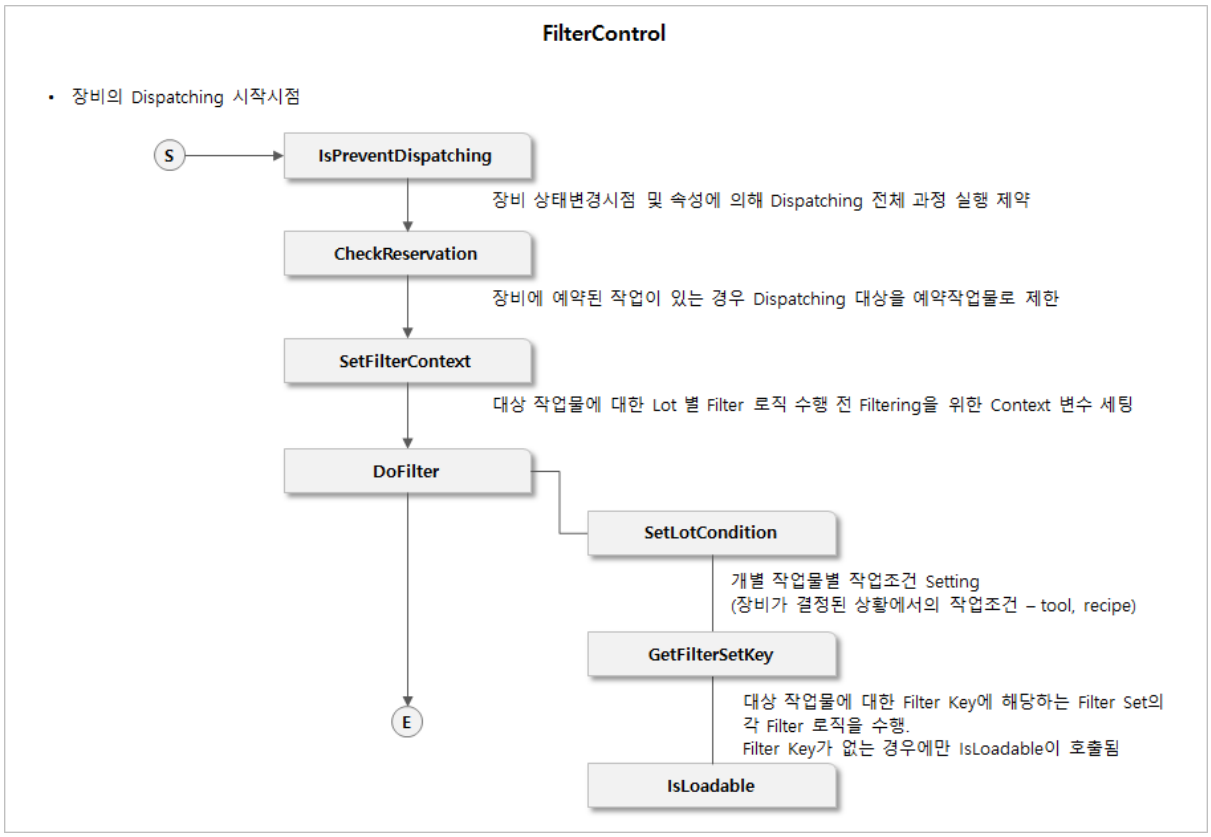
   ```
   public void NOARRANGE_HANDLER(DispatchingAgent da, IHandlingBatch hb, int destCount,
   ref bool handled)
   {
       var lot = hb.Sample as Lot;
       //로딩가능한 장비가 없는 경우 Bucketing 으로 처리
       if (destCount == 0 &&
           SemiEdsConfiguration.Instance.NoArrangeMove)
       {
           da.Factory.AddToBucketer(hb); //Bucketing 로직에서 처리방식 정의
       }
   }
   ```

9. ***GetLotGroupKey*** : Dispatching 시점에 동일 그룹으로 묶여서 그룹간의 비교를 통해 계산시간을 단축시킬 수 있는 경우 Lot 을 그룹으로 묶기 위한 Key 를 지정합니다. 예를 들어 동일 Recipe 를 사용하는 Lot 중에서는 Fifo 방식만을 사용하여 Lot 을 결정한다고 했을 때, Dispatching 시점에 어떤 Recipe 를 선택해야하는지만 결정하면 됨으로 계산시간을 많이 줄일 수 있습니다.

FilterControl 컨트롤

FilterControl 은 작업 대기 중인 장비 앞에 신규 작업물이 도착하거나, 장비가 이전 작업을 완료하고 다음 작업물을 선택하는 시점에 제어 Logic 을 담당합니다. 장비 관점의 Dispatching 제약, 작업물 관점의 Dispatching 제약을 반영할 수 있습니다.

장비에서 작업물 선택 Event 가 발생하면 다음의 FEAction 을 통해 Dispatching 진행 여부를 체크합니다.



1. ***IsPreventDispatching*** : 장비의 예약된 PM 시작 시각이 현재 시점이거나 일정 시간 이내에 들어오는 경우 혹은 그 외 장비에서 작업을 수행하는데 제약이 있는경우에 Dispatching 을 하지 않도록 합니다.

   Sample Code

```
public bool IS_PREVENT_DISPATCHING0(AoEquipment aeqp, IList<ISimEntity> entities, Ti
me waitDownTime, ref bool handled, bool prevReturnValue)
{
    // 장비 Down 시점까지 남은 시간이 10분 이내인 경우 다른 작업물을 선택하지 않음
    if (waitDownTime <= Time.FromMinutes(10))
        return true;
```

```
    return false;
}
```

2. ***CheckReservation*** : 장비의 예약 작업이 있는지 확인합니다. 예약 작업이 있는 경우, 작업물의 우선순위를 평가하는 단계를 Skip 하고 예약된 작업물 중에서 작업물을 선택하도록 사용자 Logic 을 구현할 수 있습니다.
3. ***SetFilterContext*** : 대기 중인 작업물의 개별 Filtering 조건을 평가하기 전에 모든 작업물의 평가 시 공통으로 사용될 수 있는 참고 수치를 계산하여 DispatchContext 에 저장하는 코드를 작성합니다.

> 💡 **Note**
> ***DispatchContext*** : DispatchContext 는 Dispatching 시작 시점부터 종료 시까지 각 FEAction의 구현 시 참조할 수 있는 값의 저장소와 같은 개념으로 FilterControl, DispatcherControl 의 모든 FEAction 에서 사용할 수 있습니다. 사용방식은 'Key' 를 지정하여 값을 설정하고 참조하는 방식입니다. 자세한 사용방식은 Reference Library 를 참조하시기 바랍니다.

4. ***DoFilter*** : 실제 대기 작업물에 대한 Filtering 작업을 수행하는 함수입니다. 본 함수는 Library 상의 기본 구현이 되어 있으며, 사용자가 재정의하는 경우 하위의 IsLoadable 함수는 호출되지 않습니다.

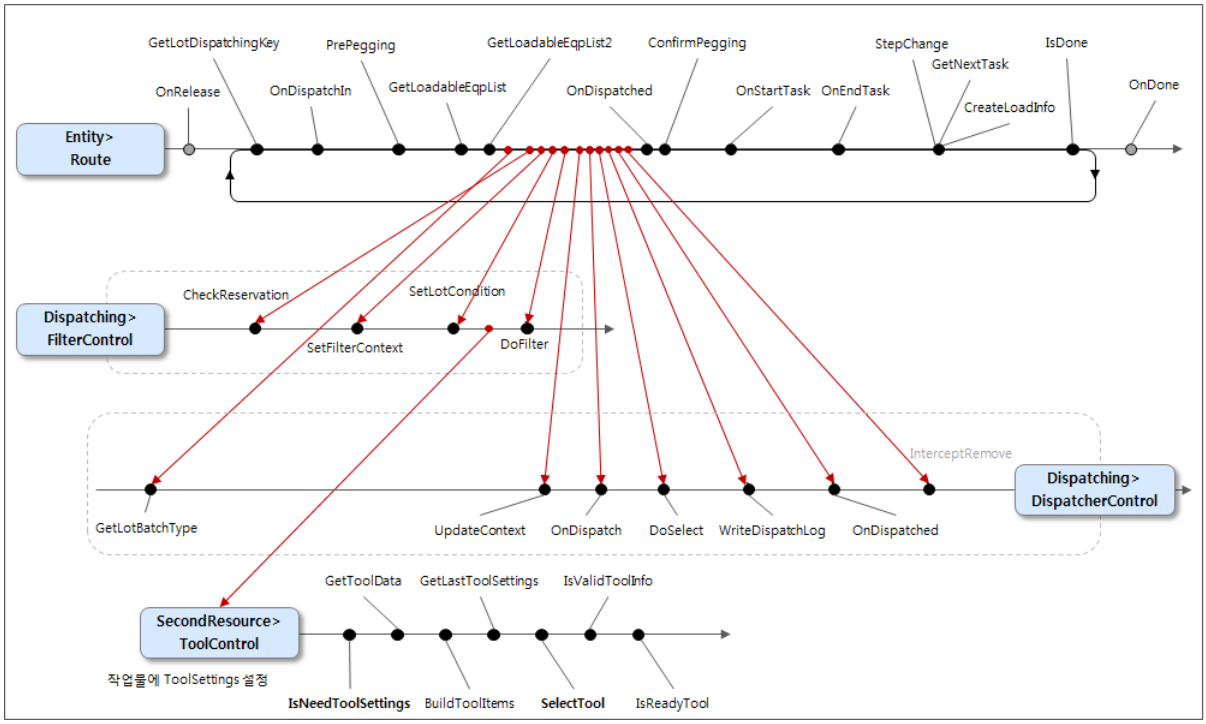
   Default Logic Example

```
public virtual IList<IHandlingBatch> DO_FILTER_DEF(AoEquipment eqp, IList<IHandlingB
atch> wips, IDispatchContext ctx, ref bool handled, IList<IHandlingBatch> prevReturn
Value)
{
 var filterControl = DispatchFilterControl.Instance;
 filterControl.SetFilterContext(eqp, wips, ctx);
 for (int i = wips.Count - 1; i >= 0; i--)
 {
     var hb = wips[i] ;
             filterControl.SetLotCondition(eqp, hb, ctx);
             if (filterControl.CheckSecondResouce(eqp, hb, ctx) == false)
             {
         wips.RemoveAt(i);
         continue;
             }
             if (filterControl.CheckSetupCrew(eqp, hb, ctx) == false)
             {
         wips.RemoveAt(i);
         continue;
             }
             if (filterControl.IsLoadable(eqp, hb, ctx) == false)
             {
         wips.RemoveAt(i);
```

```
            continue;
                }
  }
  return wips;
 }
```

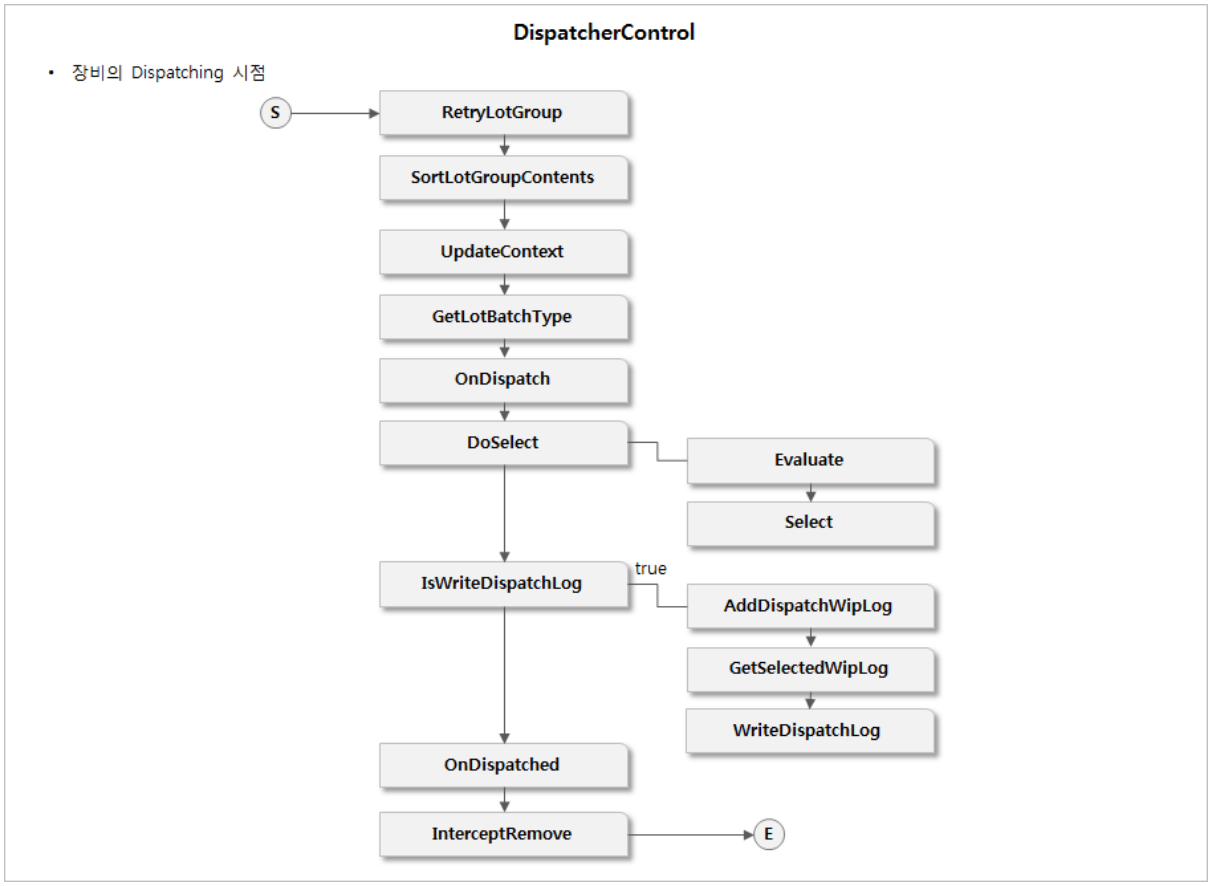
5. ***SetLotCondition*** : QueueControl Logic 처리 시 작업물의 일부 속성은 결정이 되지 않은 상태입니다. 왜냐하면 어떤 장비에서 실제 작업이 수행될 지 알 수 없기 때문입니다. 그러나 FilterControl Logic 의 판별시점에는 이미 대상 장비가 결정되어 있는 상황입니다. 따라서 작업물의 최종 작업조건이 확정 될 수 있습니다. 이러한 작업물별 최종 작업조건을 설정하는 코드를 본 FEAction을 통해 구현할 수 있습니다. 이러한 속성은 일반적으로 작업물에 설정되며, 이후 Filtering Main Logic 을 구현할 Loadable 및 DispatcherControl 의 여러 FEAction 의 구현 시 참조될 수 있습니다.
6. ***GetFilterSetKey*** : 작업물에 대해 Filtering 조건들이 여러가지 있을 수 있습니다. 본 FEAction은 FilterManager에 있는 Filter Set의 각 Filter 조건들에 의거하여 대상작업물 Filtering을 진행하기 위해 Filter Set을 찾기 위한 Filter Key를 반환해야 합니다. Filter Key가 있는 경우 IsLoadable은 호출되지 않으며 DoFilter 함수에서 Key에 해당하는 Filter Set의 각 Filter들을 호출하여 Filtering을 진행합니다. Filter Key가 없는 경우에만 IsLoadable FEAction 호출됩니다.
7. ***IsLoadable*** : 장비와 작업물의 속성에 따라 현재 시점에 작업처리가 불가한 작업물을 Filtering 하고 Processing 이 가능한 작업물만 남기는 작업을 수행하는 FEAction 입니다. 사용자는 여러가지 제약을 반영한 Loadable Logic 을 추가할 수 있습니다. 기본적인 Filter 조건이 되는 SetupCrew 의 가용여부, SecondResource 의 가용여부는 Filtering 시 반영되도록 구현이 되어 있으며, Logic 이 정상적으로 수행되기 위해서는 **Equipment > SetupControl > IsNeedSetupCrew, SecondResource > ToolControl > IsNeedToolSettings** 와 같은 함수를 구현합니다. DoFilter 함수를 사용자가 재정의 하는 경우 IsLoadable 함수가 호출되지 않습니다.

연관된 Control 들의 Action 들과 연관관계는 다음과 같습니다.



DispatcherControl 컨트롤

장비에서 가공할 다음 작업물을 찾는 시점에 장비 앞에 Filtering 되지 않고 남아있는 로딩 가능 작업물을 평가하여 우선순위를 결정하고 장비의 Processing Type 에 따라 최종 작업물을 선택하는 일련의 Logic 을 처리하는 FEComponent 입니다. 실제 시스템으로는 RTD (Real Time Dispatcher) 에 대응되는 요소이며, 다음과 같은 FEAction 으로 구성됩니다.



1. ***RetryLotGroup*** : Dispatching 전에 Lot Group 을 재구성할 수있습니다. QueueControl 의 GetLotGroupKey Action 과 연관관계가 있습니다.
2. ***SortLotGroupContents*** : LotGroup 내의 단위 작업물에 대한 우선순위 소팅을 합니다. Dispatching 시 특정 LotGroup 이 선택된 경우 해당 LotGroup 에서 가장 우선순위가 높은 Lot 을 선택하기위해 필요합니다.

   Default Logic Example

   ```
   public IList<IHandlingBatch> SORT_LOT_GROUP_CONTENTS_DEF(DispatcherBase db, IList<IH
   andlingBatch> wips, IDispatchContext ctx, ref bool handled, IList<IHandlingBatch> pr
   evReturnValue)
   {
       return db.WeightEval.SortLotGroupContents(wips, ctx);
   }
   ```

3. ***UpdateContext*** : 현재 Dispatching 대상 장비와 장비 Group 현황 (장비별 로딩제품, 대기 작업물수 등) 과 대기 작업물들의 Group 속성 (제품별 장비할당상황 등) 과 같이 작업물 단위로 계산하기보다는 일괄 계산하여 저장해두고 작업물별로 참조할 수 있는 값을 생성하여 DispatchContext 에 저장합니다. 이 작업을 통해 Dispatching 시간을 단축할 수 있습니다.

4. ***GetLotBatchType*** : 작업물이 Batch Type 작업물인 경우, 효율을 위해 Dispatching 완료시점에 Library 내부에서 HandlingBatch 를 구성하는 작업물 (Lot) 을 작업하기 전 본 Action 에서 설정한 BatchType 으로 묶어 Event 를 줄이고 시스템 효율을 올리는 용도로 사용합니다. 즉, 사용자가 정의한 Batch Lot Type 을 반환하도록 코드를 구성합니다.
5. ***OnDispatch*** : 개별 작업물의 우선순위를 평가하기 직전에 처리할 추가적인 Logic 을 구현합니다.
6. ***DoSelect*** : Dispatcher 의 Main 함수로 하위의 Evaluate, Select 함수를 호출하도록 Library 상에 구현되어있습니다. 본 함수를 재정의하게 되면 하위의 Evaluate, SelectBatch 의 실행 내용에 맞도록 사용자가 별도 Logic 을 구현해야 합니다.

   Default Logic Example

```
public IHandlingBatch[] DO_SELECT_DEF(DispatcherBase db, AoEquipment aeqp, IList<IHa
ndlingBatch> wips, IDispatchContext ctx, ref bool handled, IHandlingBatch[] returnVa
lue)
{
    var control = DispatchControl.Instance;
    if (wips.Count == 0)
        return null;
    var lotList = control.Evaluate(db, wips, ctx);
    var evalLots = new List<IHandlingBatch>((int)(lotList.Count * 1.5));
    foreach (var entity in lotList)
    {
        if (entity is LotGroup<ILot, Step>)
        {
            evalLots.AddRangeCast(entity.Contents);
        }
        else
        {
            evalLots.Add(entity);
        }
    }
    var selected = control.Select(db, aeqp, evalLots);
    if (control.IsWriteDispatchLog(aeqp))
        aeqp.Target.EqpDispatchInfo.AddDispatchInfo(evalLots, selected, aeqp.Target.
Preset);
    return selected;
}
```

7. ***Evaluate*** : 대기 중인 작업물들의 우선순위를 평가하여 Sorting 하는 함수입니다. SEEPLAN 에서 제공하는 WeightSumDispatcher, WeightSortedDispatcher 를 사용하는 경우에는 Preset 을 Input 으로 제공하고 Preset 을 구성하는 Factor Method 를 구현함으로써 별도 구현을 하지 않아도 됩니다.

   Default Logic Example

```
public IList<IHandlingBatch> EVALUATE_DEF(DispatcherBase db, IList<IHandlingBatch> w
ips, IDispatchContext ctx, ref bool handled, IList<IHandlingBatch> returnValue)
```

```
{
   if (db is FifoDispatcher)
       return wips;
   if (db.Comparer == null)
       return wips;
   return db.WeightEval.Evaluate(wips, ctx);
}
```

8. ***Select*** : Batch Processing 장비의 경우 작업물의 우선순위를 결정한 후 장비의 Processing 방식이나 공정 처리 방식에 따라 우선순위로 Sorting 된 작업물 중 작업대상을 선택합니다. 일반적으로 장비는 작업의 종류에 따라 Batch구성이 가능한 작업조건 정보를 기준정보로 관리하고 있습니다. 또한, Batch processing 장비의 경우, Processing Time 이 길어 현재 시점의 대기 재공으로 Batch 를 구성하여 작업을 시작할지 혹은 일정 시간 유입되는 작업물을 기다렸다가 더 많은 작업을 동시에 처리할지 결정하는 Logic 을 함께 구현하는 경우가 일반적입니다.

> 💡 **Note**
> Select 함수에서 복수의 작업물을 선택하는 경우, Dispatching 대상 장비가 LotBatch, BatchInline Type 의 Batch Processing 장비이면 작업물을 **[Dispatching > DispatcherControl > GetLotBatchType]** 에서 정의한 Batch 작업물로 만들어 장비에서 작업하게 됩니다. 그러나 장비가 Batch Processing Type 의 장비가 아닌 경우 선택된 작업물 중 우선순위 첫 번째 작업물은 장비에 바로 로딩되고, 후속 순위의 작업물들은 순서대로 장비의 Buffer 에 투입됩니다. 장비 Buffer 에 투입된 작업물들은 이전 작업이 완료되는 시점에 순서대로 자동로딩됩니다. Buffer 에 작업물이 있는 경우 장비는 Dispatching 을 시도하지 않습니다.

9. ***IsWriteDispatchLog*** : Dispatching Log 를 기록할지 여부를 판별하는 함수입니다. 장비별로 이를 결정할 수 있습니다. 하위의 AddDispatchWipLog, GetSelectedWipLog, WriteDispatchLog 의 세 가지 FEAction 의 경우 본 함수의 결과값이 false 인 경우 실행되지 않습니다.

10. ***AddDispatchWipLog*** : 특정 대상장비, 대상 작업물 (Lot) 에 대한 Dispatching Log 를 문자열로 만들어 반환하기 위한 함수입니다. 본 함수의 구현 결과 문자열은 WriteDispatchLog 함수에서 Parameter 로 넘어오는 EqpDispatchInfo 의 DISPATCH_INFO 속성을 설정하는데 사용합니다. 아래 예시는 Weight Preset 기반의 Dispatcher 를 사용하는 경우 장비에서 특정 작업물의 Dispatch Log 문자열을 기록하는 예제입니다.

    Default Logic Example

```
StringBuilder _log = new StringBuilder();

public string ADD_DISPATCH_WIP_LOG_DEF(Resource eqp, EntityDispatchInfo info, ILot l
ot, WeightPreset wp, ref bool handled, string returnValue)
{
    _log.Clear();

    _log.Append(lot.LotID);
    _log.AppendFormat("/{0}", lot.UnitQty);
    if (wp != null)
    {
        foreach (var factor in wp.FactorList)
        {
            var value = lot.WeightInfo.GetValue(factor);
            _log.Append("/");
            _log.Append(value);
        }
    }

    return _log.ToString();
}
```

구현 예시

```
public string ADDDISPATCHWIPLOG_0(Resource eqp, EntityDispatchInfo info, ILot lot, W
eightPreset wp, ref bool handled, string prevReturnValue)
{
    StringBuilder log = new StringBuilder();

    if (eqp.Preset != null)
    {
        foreach (var factor in eqp.Preset.FactorList)
        {
            if (string.IsNullOrEmpty(log.ToString()) == false)
                log.Append(",");

            //float s = lot.WeightInfo.GetValue(factor);
            var val = lot.WeightInfo.GetValueData(factor);
            log.Append(val.Description);
        }
    }
    SimpleMfgSemiconLot slot = lot as SimpleMfgSemiconLot;
    StringBuilder result = new StringBuilder();
    result.Append(slot.LotID);

    if (!string.IsNullOrEmpty(log.ToString()))
        result.Append(",").Append(log) ;

    return result.ToString();
}
```

상기 코드에 대한 결과는 아래와 같이 표시됩니다. 아래 결과에서 DISPATCH_INFO 에 보면 LOT ID 와 해당 LOT의 Factor 별 점수가 "," 로 구분되어 기록되어 있는 모습이 상

기 코드로 작성된 결과입니다.

LINE_ID	EQP_ID	DISPATCHING_TIME	SELECT_INFO	DISPATCH_INFO
LINE01	M01	20150324 212537	L201	L201,3.916667,1;L202,3.916667,1;L204,3.916667,1;L001,3.9
LINE01	M02	20150324 212537	L202	L202,3.916667,1;L203,3.916667,1;L205,3.916667,1;L001,3.9
LINE01	C01	20150324 212537	L208	L208,4,8
LINE01	C01	20150324 222537	L003	L003,0.9166667,1;L002,0.9166667,1
LINE01	C01	20150324 232537	L002	L002,1.916667,1

11. ***GetSelectedWipLog*** : Dispatching 결과 선택된 작업물에 대한 Log 문자열을 반환합니다. 함수의 Parameter 로 선택된 작업물 정보가 입력되면 문자열을 만들어 반환합니다. 상기 결과 예제에서 SELECT_INFO 에 표시되는 문자열입니다.

    Default Logic Example

```
public string GET_SELECTED_WIP_LOG_DEF(Resource resource, IHandlingBatch[] sels, ref
bool handled, string prevReturnValue)
{
    _log.Clear();
    foreach (var hb in sels)
    {
        if (_log.Length > 0)
            _log.Append(";");
        var lotID = hb.Sample.LotID;
        _log.Append(lotID);
    }
    return _log.ToString();
}
```

12. ***WriteDispatchLog*** : Dispatching 결과를 원하는 형태의 Output Schema에 따라 출력하는 코드를 작성합니다. 상기 결과 Output 을 출력하는 코드를 작성합니다.

    구현 예시

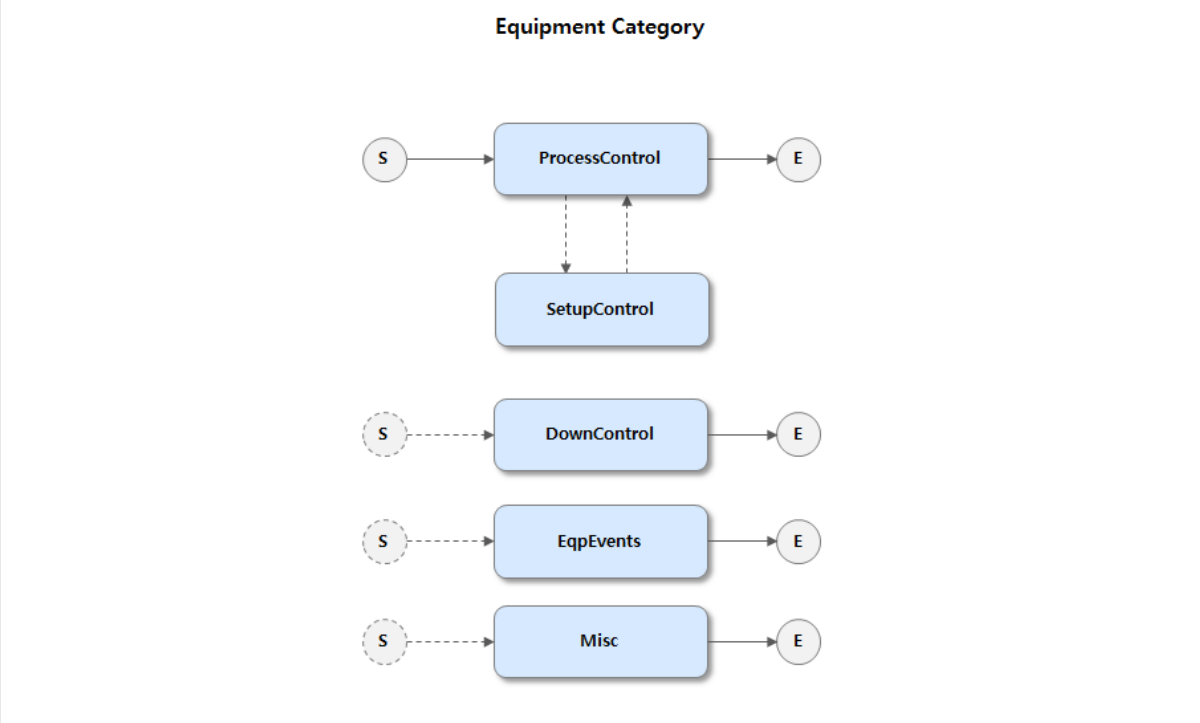
```
public void WRITEDISPATCHLOG_0(DispatchingAgent da, Mozart.SeePlan.DataModel.EqpDisp
atchInfo info, ref bool handled)
{
    Outputs.EqpDispatchInfo dispatchlog = new Outputs.EqpDispatchInfo();

    dispatchlog.LINE_ID= (info.TargetEqp as SimpleMfgSemiconEqp).LineID ;
    dispatchlog.EQP_ID = (info.TargetEqp as SimpleMfgSemiconEqp).ResID;
    dispatchlog.DISPATCHING_TIME = info.DispatchTime.DbToString(true);
    dispatchlog.DISPATCH_INFO = info.DispatchWipLog;
    dispatchlog.FILTER_INFO = info.FilteredWipLog;
    dispatchlog.SELECT_INFO = info.SelectedWipLog;
    OutputMart.Instance.EqpDispatchInfo.Add(dispatchlog);
}
```

13. ***OnDispatched*** : 작업물이 선택된 직후 작업물 혹은 장비속성의 변경이 필요할 때 이를 구현합니다. 장비 입장에서는 다음 작업할 작업물이 확정된 상태임으로 이 시점에 필요한 여러 정보를 업데이트 할 수 있습니다.
14. ***InterceptRemove*** : 작업물이 선택되어 Dispatching Queue 에서 제거되는 시점에 호출되며, Queue 에서 삭제하는 작업을 처리하지 않고 바로 작업물을 반환해야 하는 경우 해당 Logic 을 구현합니다.

Equipment

Equipment Category 는 작업물의 Processing 을 담당하는 대표적인 자원, 장비의 제어를 위한 DownControl, SetupControl, ProcessControl 및 장비 Event 및 기타 장비에 대한 Logic 처리를 위한 5가지 Control 로 구성됩니다.

아래 그림은 Equipment Category 의 FEComponent 구성 및 타 영역과의 관계를 도식화하여 설명합니다.



Equipment Category 의 각 Control 은 다음과 같은 역할을 수행합니다.

ProcessControl
작업물이 장비에 로딩되는 시점부터 Setup, 가공, 언로딩 될 때까지의 모든 제어 Logic 을 구현하기 위한 FEComponent 입니다. 본 컨트롤의 Setup 판별 및 Setup 처리를 위해 SetupControl 과 연계하게 됩니다.

SetupControl
장비 Setup 여부판별 및 실행 제어를 위한 FEComponent 입니다. Process Control 로 부터 시작하여 제어 종료 후 ProcessControl 로 제어 흐름을 넘겨주도록 구성되어 있습니다.

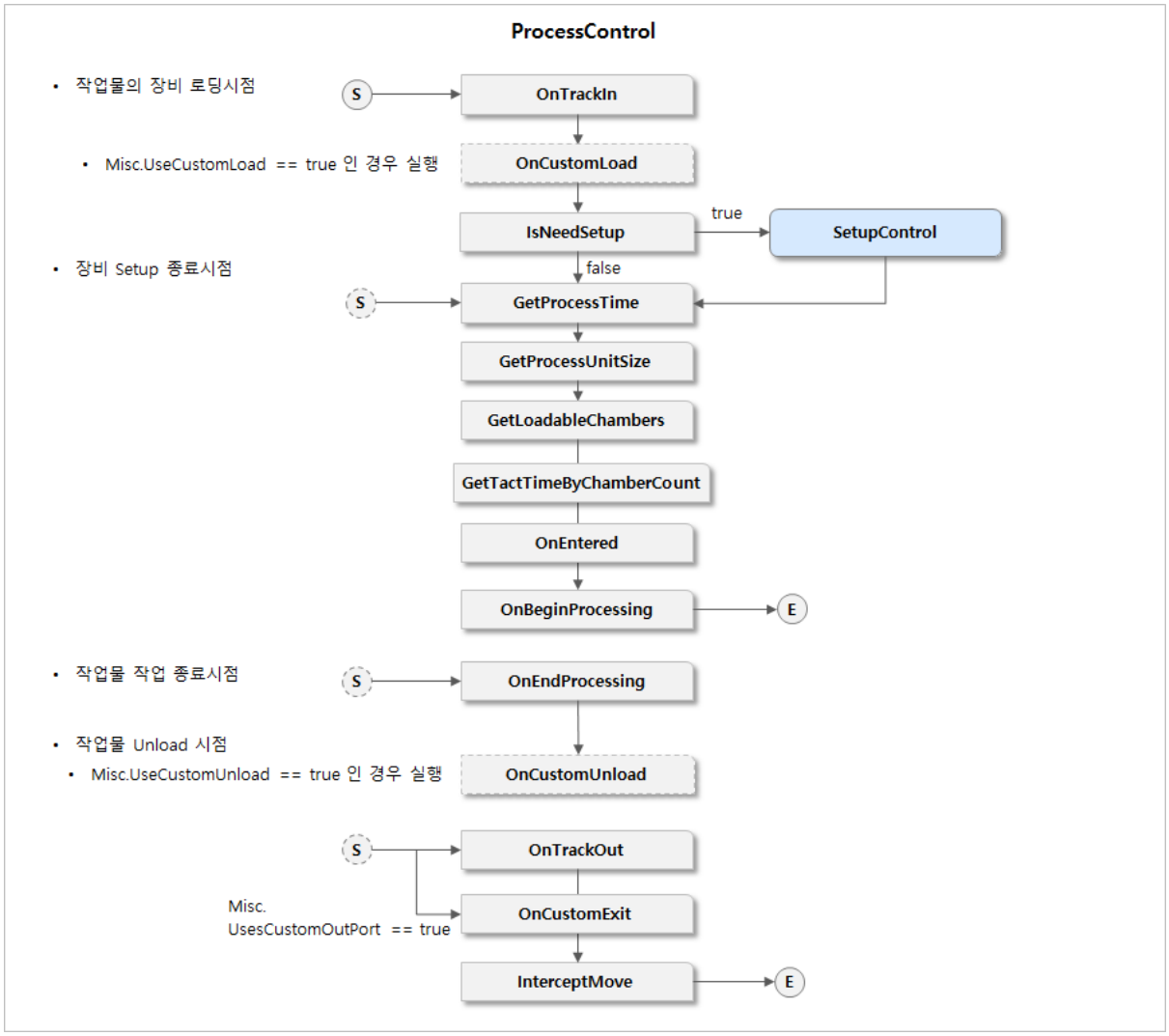
DownControl
장비의 Down (PM, Failure) 상태 제어를 위한 FEComponent 입니다.

EqpEvents
상기 Main Logic 을 제외한 장비에서 발생할 수 있는 Event 로 사용자가 해당 Event 시점에 사용자 정의 Logic 을 구현할 수 있습니다.

Misc
장비의 상태 및 특성을 설정/확인 하기 위해 불특정 시점에 호출될 수 있는 함수의 집합입니다.

ProcessControl 컨트롤

작업물이 장비에 로딩되는 시점부터 Processing 후 Unloading 될 때까지의 모든 과정을 컨트롤하기 위한 FEComponent 입니다. FEAction 은 아래 그림과 같이 크게 4부분으로 나눠집니다.



작업물의 장비 로딩시점

1. ***OnTrackIn*** : 작업물의 장비 로딩이 끝나고 Processing 시작 직전에 실행되는 함수로 이 시점에 처리가 필요한 Logic 을 구현합니다. 작업물이 장비에 Load 될 때 수행하는 기본작업 및 Data 수집은 이미 완료된 상태이며, 추가적인 사용자 처리 Logic 을 구현할 때 사용할 수 있습니다.
2. ***OnCustomLoad*** : 작업물 로딩 시점에 작업물의 속성을 변경한다거나, 작업물을 Split 하여 연속 진행하게 한다던지 하는 사용자 Logic 을 구현합니다. 본 Action 은 [Misc.UseCustomLoad]() 함수의 값이 true 인 경우에만 동작합니다.
3. ***IsNeedSetup*** : 장비에 들어온 작업물의 Setup 필요 여부를 판별합니다. Setup 이 필요한 경우, 제어권을 SetupContol 에서 받아 진행합니다.

장비 Setup 종료시점 (작업준비시점)

1. ***GetProcessTime*** : 작업물의 Processing Time 을 설정합니다. 결과로 ProcTimeInfo 클래스를 반환하며 Process Time 과 Tact Time으로 구성되어 있습니다. Processing 시작 전에 작업물의 현재 공정을 기준으로 현 장비에서의 Process Time을 설정해두면, Processing 시점에 이 정보를 활용하여 Logic 을 처리하도록 Simulation Model 이 구성되어 있습니다.

> 💡 **Note**
> 기본적으로 Lot 을 구성하는 개별 Unit 당 소요되는 Process Time, Tact Time 으로 설정해야 합니다. 만일 사용자 조직의 Data 가 이와 맞지 않는 경우 Input 을 가공하여 Library 에서 요구하는 단위의 Data 로 입력해주어야 합니다.

구현 예시

```
public ProcTimeInfo GETPROCESSTIME(Mozart.SePlan.Simulation.AoEquipment aeqp, IHand
lingBatch hb, ref bool handled, ProcTimeInfo prevReturnValue)
{
    SimpleMfgSemiconLot lot = SimHelper.GetLot(hb);
    ProcTimeInfo result = new ProcTimeInfo();
    // 기본시간 설정
    result.FlowTime = TimeSpan.FromMinutes(60);
    result.TactTime = TimeSpan.FromMinutes(60);

    // 시간 검색
    string recipe = lot.Product.ProductID + "." + lot.CurrentStep.StepID;
    object[] keys = new object[] { lot.Product.ProductID, lot.Process.ProcessID, lo
t.CurrentStep.StepID, recipe, aeqp.EqpID };
    Inputs.StepTime st = InputMart.Instance.StepTime.Rows.Find(keys);

    if (st != null)
    {
        result.FlowTime = TimeSpan.FromSeconds(st.PROC_TIME);
        result.TactTime = TimeSpan.FromSeconds(st.TACT_TIME);
    }

    return result;
}
```

2. ***GetProcessUnitSize*** : 작업물의 Processing Unit Size 를 반환합니다. Processing Unit Size 는 작업물을 구성하는 최소단위 Unit 의 수를 의미합니다. 기준정보의 구성과 장비에서 Processing Time 을 계산하는 방식에 따라 Processing Unit 의 처리방식이 변경될 수 있습니다. 다만, Batch 장비의 경우에는 Processing 시간이 작업물의 실제 Unit 수량과 상관없이 실적 집계에서 사용하는 기본 Unit Size 를 사용함에 따라 이를 Logic 에 반영해 주어야 합니다.

> **Note**
> 실제 장비에서의 Tact Time, Process Time 은 Handling Batch 단위로 정해집니다. 즉 Batch Type 으로 여러 Lot 이 묶여 투입되더라도 장비 내에서 Processing 될 때는 모두 단일 작업물과 같이 처리되는 방식입니다. 따라서, 작업물의 가공 작업 시간은 아래와 같이 설정됩니다. Batch 장비에서의 Unit Processing Time 의 경우 기본적으로 1 Lot 의 작업 소요시간과 배치로 묶이는 n Lot 작업시간이 같아 Unit 당 작업시간은 작업물을 구성하는 평균적인 Unit Size 를 통해 통계치를 수집하게 됨으로 아래의 예시와 같이 실제 작업배치의 가공시간을 산출하는 경우에도 평균 Unit Size 를 사용하여 가공시간을 계산하도록 합니다.

Flow Time = { Unit Proc Time } * { Process Unit Size }

Tact Time = { Unit Tact Time } * { Process Unit Size }

Default Logic Example

```
public double GET_PROCESS_UNIT_SIZE_DEF(AoEquipment aeqp, IHandlingBatch hb, ref boo
l handled, double prevReturnValue)
{
   double unitSize = SeeplanConfiguration.Instance.LotUnitSize;
   if (!aeqp.IsBatchType())
       unitSize = hb.UnitQty;
   return unitSize;
}
```

구현 예시

```
public double GETPROCESSUNITSIZE_0(AoEquipment aeqp, IHandlingBatch hb, ref bool han
dled, double prevReturnValue)
{
    // 기본 Lot Unit size 정보를 기본 설정
    double unitSize = SeeplanConfiguration.Instance.LotUnitSize;
    SimpleMfgSemiconLot lot = SimHelper.GetLot(hb);
    double unitBatchSize = 13 ;

    if (aeqp.Target.SimType == SimEqpType.UnitBatch)
    {
        // UnitBatch 인 경우 UnitBatch 의 배수에 해당하는 만큼 시간 소요
        unitSize = Math.Ceiling(((double)lot.UnitQty / unitBatchSize)) * unitBatchSi
ze;
    }
    else if (!aeqp.IsBatchType())
    {
        // Batch Type 이 아닌 경우에는 Lot 의 실 Unit 수량만큼 시간 소요
        unitSize = lot.UnitQty;
    }
}
```

3. ***GetLoadableChambers*** : SimType 이 ParallelChamber 인 장비에서 장비 로딩시점에 Chamber 의 상태와 Chamber 단위의 로딩 가능 정보를 기반으로 실제 로딩 가능한 Chamber 정보만 반환하는 함수입니다.

   Default Logic Example

```
public string[] GET_LOADABLE_CHAMBERS_DEF(AoChamberProc2 cproc, IHandlingBatch hb, r
ef bool handled, string[] prevReturnValue)
 {
     // default values : all available chambers
     DateTime now = cproc.NowDT;
     List<string> chambers = new List<string>();
     foreach (var info in cproc.Chambers)
     {
         if (info.GetAvailableTime() <= now)
             chambers.Add(info.Label);
     }
     return chambers.ToArray();
 }
```

   구현 예시

```
public string[] GETLOADABLECHAMBERS_0(AoChamberProc2 cproc, ILot lot, ref bool handl
ed, string[] prevReturnValue)
{
  SimpleMfgSemiconLot slot = lot as SimpleMfgSemiconLot ;
  string[] chambers = slot.CurrentPlan.LoadableChambers;

  List<string> loadables = new List<string>() ;
  foreach (AoChamberProc2.ChamberInfo info in cproc.Chambers)
  {
      foreach (string id in chambers)
      {
          if (id == info.Label)
          {
              if (info.GetAvailableTime() <= cproc.NowDT)
                  loadables.Add(id);
              break;
          }
      }
  }

  return loadables.ToArray();
}
```

4. ***GetTactTimeByChamberCount***: 설비 타입이 Parellel Chamber인 경우에만 호출되며 설비에 작업물이 로드될 때 사용할 챔버 수에 따라 TactTime 계산하고 반환하는 FEAction입니다.

5. ***OnEntered*** : 작업물이 장비에 들어오고 Processing 을 시작하기 직전에 호출됩니다. 작업완료 시간전에 처리가 필요한 로직을 구현할 수 있습니다.
6. ***OnBeginProcessing*** : 작업물의 Processing 을 시작하기 직전에 호출되며, 해당 시점에 필요한 Logic 을 구현할 수 있습니다.

작업종료 시점

1. ***OnEndProcessing*** : 장비에서 작업물의 작업을 마친 시점에 처리할 Logic 을 구현합니다. Processing 이 완료되고 아직 Unload 되기 전 상태에서 작업물 속성 및 통계치, 제약조건에 대한 속성을 변경할 수 있습니다.

Unloading 시점

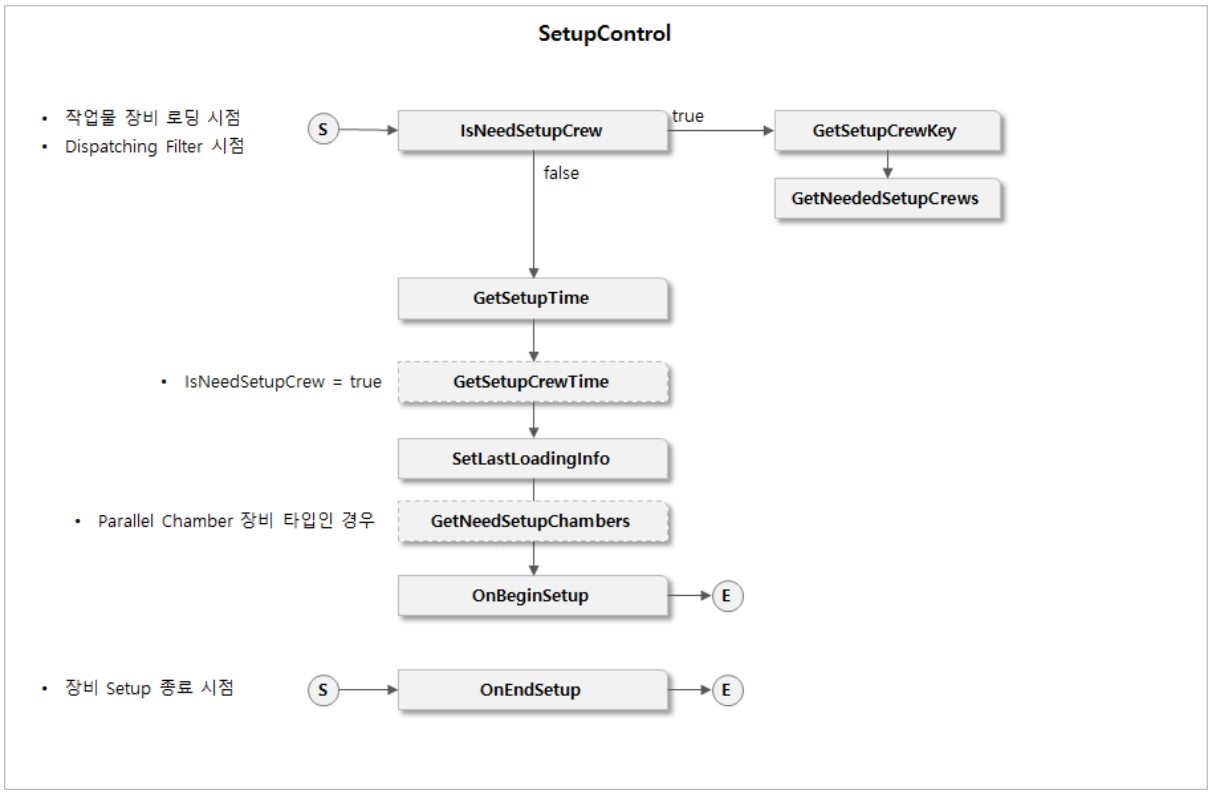
1. ***OnTrackOut*** : 장비에서 작업물이 Track Out 되는 시점에 처리할 Logic 을 구현합니다. 작업물의 현 공정 정보를 다음 공정으로 설정하기 직전 마지막 Logic 구현 지점입니다.
2. ***OnCustomExit*** : Misc.UseCustomOutPort 함수를 통해 사용자 정의형 Output Port 를 사용하도록 설정한 경우 작업물을 정상적인 Unloader 를 경유하지 않고 사용자 정의 Port 를 통해 장비를 빠져나가게 됩니다. 작업 종료 후 장비에서 Unload 되는 시점에서 처리해주어야 하는 작업을 사용자가 모두 구현해주어야 합니다.
3. ***InterceptMove*** : 작업물이 장비에서 최종 Track Out 되어 Transfer 로 넘어가기 전에 작업물의 Split/Merge 등의 작업을 수행할 수 있는 FEAction 입니다.
4. ***OnCustomUnload***: 작업물이 장비에서 Unloading되기 전에 설비 혹은 작업물의 속성을 업데이트 할 수 있습니다. 예를 들어 작업물을 Unloading 할 때 Unloading 시간을 별도로 모델링해야 하너간, Unload 시 사용하는 도구나 자원이 별도로 존재하여 이들 자원을 모델링 할 필요가 있는 경우 본 FEAction에 이러한 로직을 구현 할 수 있습니다. OnCustomUnload는 **Simulation > Equipment > Misc > UseCustomUnload** 함수의 반환값이 `true` 인 경우 호출되며 반환 값이 `false` 이면 OnCustomUnload 대신 OnTrackOut 호출됩니다.

SetupControl 컨트롤

작업물이 Processing 되기 위해서 장비는 작업물을 가공할 수 있도록 Setting 되어야 합니다. 만일 현재 작업을 수행할 수 있는 조건을 만족하지 못하는 상태라면 장비 혹은 작업물의 설정

을 변경하여 작업을 할 수 있도록 만들어야 하는데 이러한 일련의 과정을 Setup 이라 표현합니다. Setup 의 발생과 처리과정을 모델링하고 컨트롤하기 위해서는 Setup 이 필요한지, Setup 을 위한 추가적인 자원이 필요한지, 시간은 얼마나 소요되는지 등의 정보를 결정하고 Simulator 에 제공해주어야 합니다.

EqpSetupControl 은 Setup 과 관련된 일련의 제어를 위해 다음의 FEAction 을 제공합니다.



1. ***IsNeedSetupCrew*** : 장비 Setup 을 위해 Setup Crew 가 필요한지 여부를 판별하는 Logic 을 구현합니다. 실제 대상 시스템에서 Setup 에 필요한 모든 2차적인 Resource 를 대상으로 하는 것은 아니며 Setup 을 위한 Resource 의 Capacity 가 유한하여 사용에 제약이 가해지는 경우만 대상으로 합니다.
2. ***GetSetupCrewKey*** : Setup 을 위한 Setup Crew 가 필요한 경우, 작업물의 조건에 따라 어떤 Setup Crew 가 필요한지 해당 Crew 를 찾기 위한 Key 정보를 반환하는 Logic 을 구현합니다.

   Default Logic Example

```
public string GET_SETUP_CREW_KEY_DEF(AoEquipment aeqp, IHandlingBatch hb, ref bool h
andled, string prevReturnValue)
{
   return aeqp.EqpID;
}
```

3. ***GetNeededSetupCrews*** : Setup에 필요한 Setup Resource(SetupCrew) 양(수)을 산출하는 Logic 을 구현합니다.
4. ***GetSetupTime*** : Setup 을 위한 모든 조건이 만족한 경우 해당 작업의 Setup 소요시간을 산출합니다. 작업물과 장비와 관계없이 고정적인 시간을 사용할 수 있으나, Setup 종류에 따라 시간의 차이가 많이 나는 경우 이에 대한 Model 이 반영되어야 합니다.
5. ***GetSetupCrewTime***: 설비의 Setup 소요 시간 중 Setup Crew가 실질적으로 필요로 하는 시간을 반환하는 로직을 구현하는 FEAction입니다. 반환된 시간이 지나면 대상 설비의 Setup이 완료되지 않더라도 대상 설비에서 Setup Crew를 해제하고 Setup이 예정되어 있는 다른 설비에 할당합니다.
6. ***SetLastLoadingInfo*** : 장비에서 마지막으로 작업중인 작업물 정보를 Update 합니다. 경우에 따라서는 Setup 이 작업 수행시점마다 발생할 수 있으나 일반적으로는 장비에 Setup 된 조건과 동일한 작업물을 계속 진행하여 Setup Loss 를 줄이려고 합니다. 따라서 Setup 발생 시점에 마지막 Setup 된 작업조건을 기록하는 작업을 해야합니다. Setup 에 영향을 미치는 작업조건은 공정 혹은 제조업종별로 차이가 날 수 있음으로 이 부분은 사용자가 정의할 수 있습니다. 여기서 설정된 마지막 작업조건 정보는 1,2 번의 Setup 판별 및 Setup Resource 의 필요여부 판별 시 사용될 수 있습니다.

   Default Logic Example

```
public LoadInfo SET_LAST_LOADING_INFO_DEF(AoEquipment aeqp, IHandlingBatch hb, ref b
ool handled, LoadInfo prevReturnValue)
{
   var lot = hb.Sample;
   return lot.CurrentPlan;
}
```

7. ***GetNeedSetupChambers***: Parallel Chamber 설비 타입에서 각 챔버 별 Setup을 적용하기 위한 FEAction입니다. 본 함수에서 반환된 목록의 챔버에만 Setup이 발생하며, 각 챔버 별 Setup 시간을 개별적으로 적용하고자 하는 경우 ChamberInfo.SetSetupTime(Time) 함수를 사용하면 됩니다.

   Sample Code

```
public ISet<string> GET_NEED_SETUP_CHAMBERS0(Mozart.SeePlan.Simulation.AoEquipment a
eqp, ChamberInfo[] loadableChambers, IHandlingBatch hb, ref bool handled, ISet<strin
g> prevReturnValue)
{
    SiteEqp eqp = aeqp.Target as SiteEqp;
    SiteLot lot = hb.Sample as SiteLot;
    HashSet<string> list = new HashSet<string>();
    if (eqp.SimType == Mozart.SePlan.DataModel.SimEqpType.ParallelChamber)
    {
        foreach (ChamberInfo chamber in loadableChambers)
        {
```

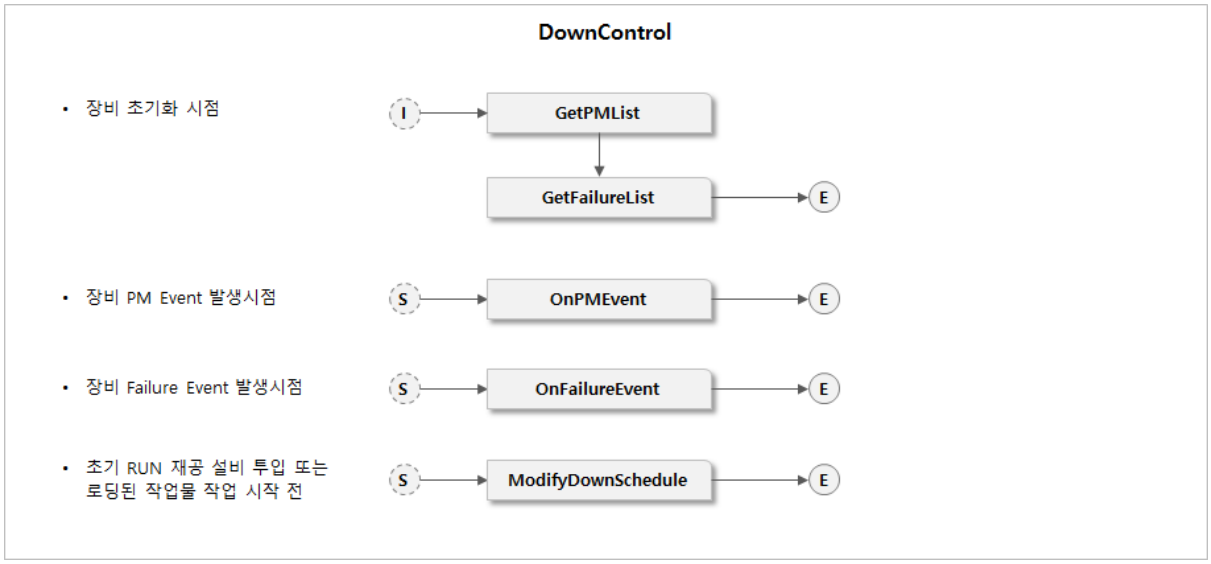
```
            string subEqpID = chamber.Label;
            if (string.IsNullOrEmpty(subEqpID))
                continue;
            Time setupTime = Time.FromMinutes(30d);
            chamber.SetSetupTime(setupTime);
            list.Add(subEqpID);
        }
    }
    return list;
}
```

8. ***OnBeginSetup*** : Setup 을 시작하는 시점에 호출됩니다. Library 상에서 Setup 시작 시 처리할 기본 Logic 을 수행한 후 호출됩니다.
9. ***OnEndSetup*** : Setup 을 종료하는 시점에 호출됩니다. Library 상에서 Setup 종료 시 처리할 기본 Logic 을 수행한 후 호출됩니다.

DownControl 컨트롤

Simulation 중 장비상태가 Down 상태로 변경 될 수 있는 요인을 모델링하여 실제 제조라인에서 발생 할 수 있는 장비의 비가동 상황을 Simulation 에서 반영하도록 지원하는 컨트롤 Model 입니다. 장비 Down 은 PM (Preventive Maintenance) 과 같이 예측 가능한 경우와 Failure 와 같이 예측이 불가능한 경우가 있습니다. 반도체 제조업과 같은 장비집약적 제조업의 경우 PM 를 계획적으로 수행하는 경우가 많아 Simulation 에서도 PM Schedule 을 반영하는 경우가 일반적입니다. Failure 의 경우 확률적인 빈도와 MTTR (Mean Time To Repair) 을 반영하여 Failure 자체를 모델링하는 경우도 있으나 평균 가동율을 장비 Processing Time 에 반영해서 장비의 Capacity 를 조정하는 방법을 사용하기도 합니다. 이 경우에는 Failure 에 대한 별도의 모델링이 불필요하게 됩니다.

EqpDownControl 은 장비 Down 에 대한 Logic 제어를 위해 다음의 FEAction 을 제합니다.



1. ***GetPMList*** : PM Schedule 을 기준으로 PM Events 를 생성하고 List 를 반환하도록 구현합니다. 모델링 대상 사이트의 PM Schedule 을 기준으로 PM 시작 및 종료 Event 를 생성하는 것이 일반적입니다.
2. ***GetFailureList*** : Failure 처리 Rule 에 따라 Failure Events 를 생성하고 List 를 반환하도록 구현합니다. 장비 고장 행태가 확률적으로 유의미한 수준으로 발생하는 경우에 반영할 수 있습니다.
3. ***OnFailureEvent*** : 장비에 등록된 Failure Event 발생 시 장비의 상태를 변경하고 추가적인 장비 로딩을 할 수 없도록 처리합니다.

   Default Definition

   ```
   public void ON_FAILURE_EVENT_DEF(AoEquipment aeqp, FailureSchedule fs, DownEventType
   det, ref bool handled)
   {
       if (det == DownEventType.End)
       {
           aeqp.Loader.Unblock();
           aeqp.WriteHistoryAfterBreak();
       }
       else
       {
           aeqp.WriteHistory(LoadingStates.DOWN);
           aeqp.Loader.Block();
       }
   }
   ```

4. ***OnPMEvent*** : 장비에 등록된 PM Event 발생 시 장비의 상태를 변경하고 추가적인 장비 로딩을 할 수 없도록 처리합니다.

> **Note**
> PM 과 Failure Event 발생 시 장비에서의 처리방식은 기본적으로 동일합니다. 다만, 장비의 상태 코드가 다르며, Event 종류에 따라 사용자가 정의한 별도의 Logic 을 추가할 수 있습니다.

Default Definition

```
public void ON_PM_EVENT_DEF(AoEquipment aeqp, PMSchedule fs, DownEventType det, ref
 bool handled)
{
    if (aeqp.SetParallelChamberPM(fs, det))
        return;
    if (det == DownEventType.Start)
    {
        aeqp.Loader.Block();
        aeqp.WriteHistory(LoadingStates.PM);
    }
    else
    {
        aeqp.Loader.Unblock();
        aeqp.WriteHistoryAfterBreak();
        aeqp.SetModified();
    }
}
```

5. ***ModifyDownSchedule***: 설비에서 작업물 Processing 중에 작업 완료 시간 전에 Down(PM, Failure) 등의 Event 가 존재하는 경우 해당 Schedule 을 지연시키거나 취소하는 등의 사용자 로직을 처리합니다. 본 FEAction은 재공 초기화 시점에 초기 재공을 설비에 추가하는 **Entity > WipInit > LocateForRun**의 LOCATE_FOR_RUN_DEF Default Definition에서 호출 그리고 시뮬레이션 중 재공이 설비에서 작업물이 로딩되고 Processing이 시작되기 직전에 호출됩니다.

Default Definition

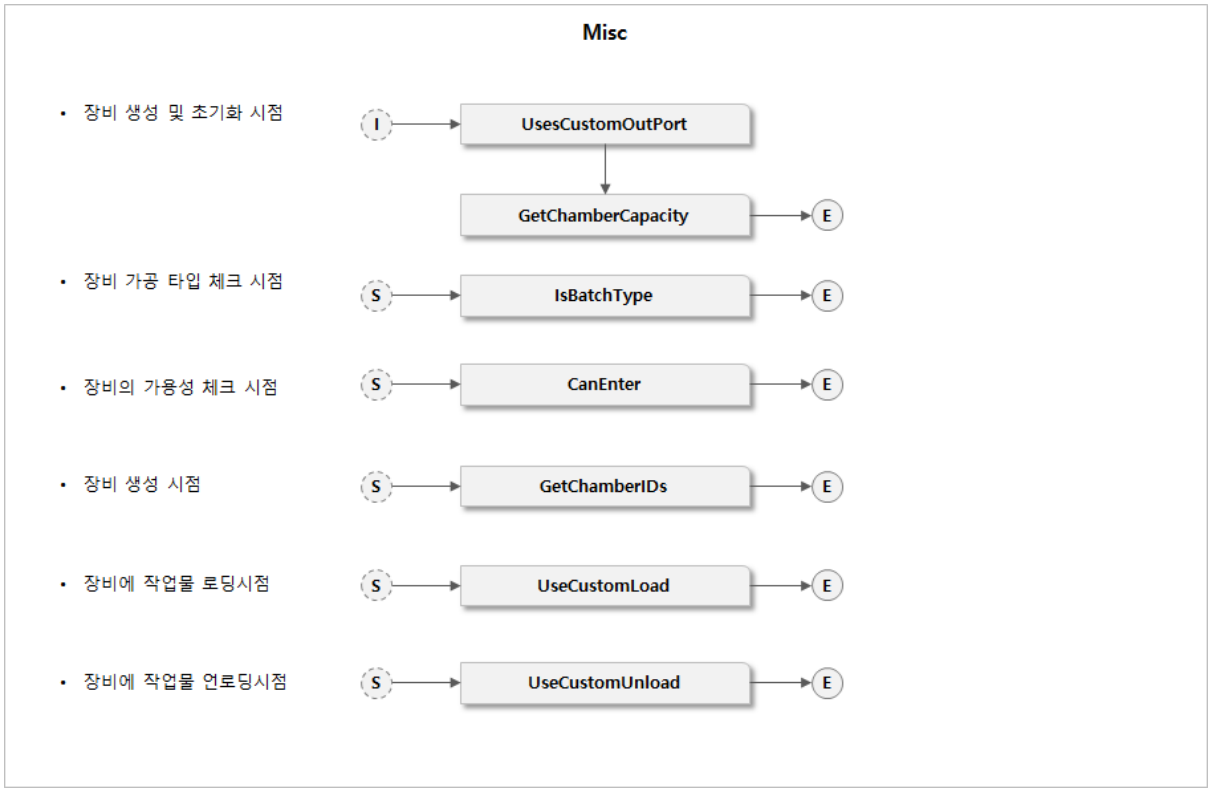
```
public void MODIFY_DOWN_SCHEDULE_DEF(AoEquipment aeqp, ref bool handled)
{
    var downManager = aeqp.DownManager;
    if (downManager == null)
        return;
    DateTime now = aeqp.NowDT;
    var startItems = downManager.GetStartScheduleItems(Time.MaxValue);
    var lastEndTime = Time.MinValue;
    var ignoreBlock = new List<string>();
    foreach (var item in startItems)
    {
        var schedule = item.Tag as PeriodSection;
        var rule = schedule.ScheduleType;
        if (rule != DownScheduleType.ShiftBackward
```

```
            && rule != DownScheduleType.ShiftBackwardStartTimeOnly
            && rule != DownScheduleType.Cancel)
            continue;

        var componentID = string.Empty;
        var isPM = schedule is PMSchedule;
        if (isPM)
        {
            var pm = schedule as PMSchedule;
            if (pm.PMType == PMType.Component)
                componentID = pm.ComponentID;
        }

        if (!ignoreBlock.Contains(componentID) && aeqp.IsBlocked(componentID))
            continue;

        var remainEndTime = aeqp.GetRemainTimeToEnd(componentID);
        if (lastEndTime > remainEndTime)
            remainEndTime = lastEndTime;
        var wait = item.EventTime - now;
        if (remainEndTime != Time.MaxValue && wait >= Time.Zero && wait < remainEndTime)
        {
            var adjStartTime = now.AddSeconds(remainEndTime.TotalSeconds);
            var adjEndTime = DateTime.MaxValue;
            var adjSchedule = (PeriodSection)schedule.Clone();
            var adjusted = false;
            switch (rule)
            {
                case DownScheduleType.ShiftBackward:
                    adjEndTime = adjStartTime.Add(schedule.Duration);
                    adjSchedule.StartTime = adjStartTime;
                    adjSchedule.EndTime = adjEndTime;

                    downManager.AdjustEvent(item, adjSchedule);
                    adjusted = true;
                    lastEndTime = adjEndTime - now;
                    break;

                case DownScheduleType.ShiftBackwardStartTimeOnly:
                    var newDownDuration = schedule.EndTime - adjStartTime;

                    downManager.CancelEvent(item);

                    if (newDownDuration <= Time.Zero)
                        continue;

                    adjusted = true;
                    adjSchedule.StartTime = adjStartTime;
                    downManager.AddEvent(adjSchedule);
                    lastEndTime = adjSchedule.EndTime - now;
                    break;

                case DownScheduleType.Cancel:
                    downManager.CancelEvent(item);
                    continue;
            }
            if (adjusted && adjSchedule is PMSchedule && aeqp.IsParallelChamber)
            {
                var pm = adjSchedule as PMSchedule;
```

```
                if (pm.PMType == PMType.Full)
                    aeqp.Loader.Block();
                else
                {
                    var cproc = aeqp.Processes[0] as AoChamberProc2;
                    var block = true;
                    foreach (var c in cproc.Chambers)
                    {
                        if (c.Label == componentID)
                        {
                            if (c.BlockEndTime < now && !ignoreBlock.Contains(componentI
D))
                                ignoreBlock.Add(componentID);
                            c.BlockEndTime = pm.EndTime;
                        }
                        else if (c.Active)
                            block = false;
                    }
                    if (block)
                        aeqp.Loader.Block();
                }
            }
            else
                aeqp.Loader.Block();
        }
    }
}
```

Misc 컨트롤

장비의 상태 및 특성을 설정/확인 하기 위해 불특정 시점에 호출될 수 있는 함수의 집합입니다.



1. ***UseCustomOutPort*** : 장비의 Output Port 에서 작업물의 완료 (Unload) Even에 대한 사용자 정의 Logic 을 사용할 지 여부를 설정합니다. True 를 설정하는 경우에는 [ProcessControl.OnCustomExit](#) FEAction 을 구현해주어야 합니다. SEEPLAN Library 의 경우 특별한 설정이 없는 경우 장비에서 Processing 이 끝난 작업물을 AoUnloader 를 통해 내보내도록 되어있으며, 사용자 정의 Logic 을 구현하는 경우에만 본 함수를 구현합니다.
2. ***GetChamberCapacity*** : SimEqpType = Chamber 장비에 설정이 필요한 속성을 정의하기 위한 FEAction 입니다. Chamber 타입 장비의 Chamber 수를 설정합니다.
3. ***IsBatchType*** : 장비가 여러 개의 작업물을 Batch 로 처리하는 타입의 장비인지 여부를 반환합니다. 기본적으로 장비의 SimEqpType 에 따라 BatchInline, LotBatch 의 SimType 만 BatchType 으로 구분되며, Library 에서 기본으로 두 가지 타입에 대해 BatchType 으로 분류하도록 되어있습니다. 다만, 초기 입력 Data 를 기준으로 특정장비의 SimType 을 별도 설정하거나 새로운 타입을 만드는 경우 해당 Action 을 구현해주어야 합니다.

   Default code sample

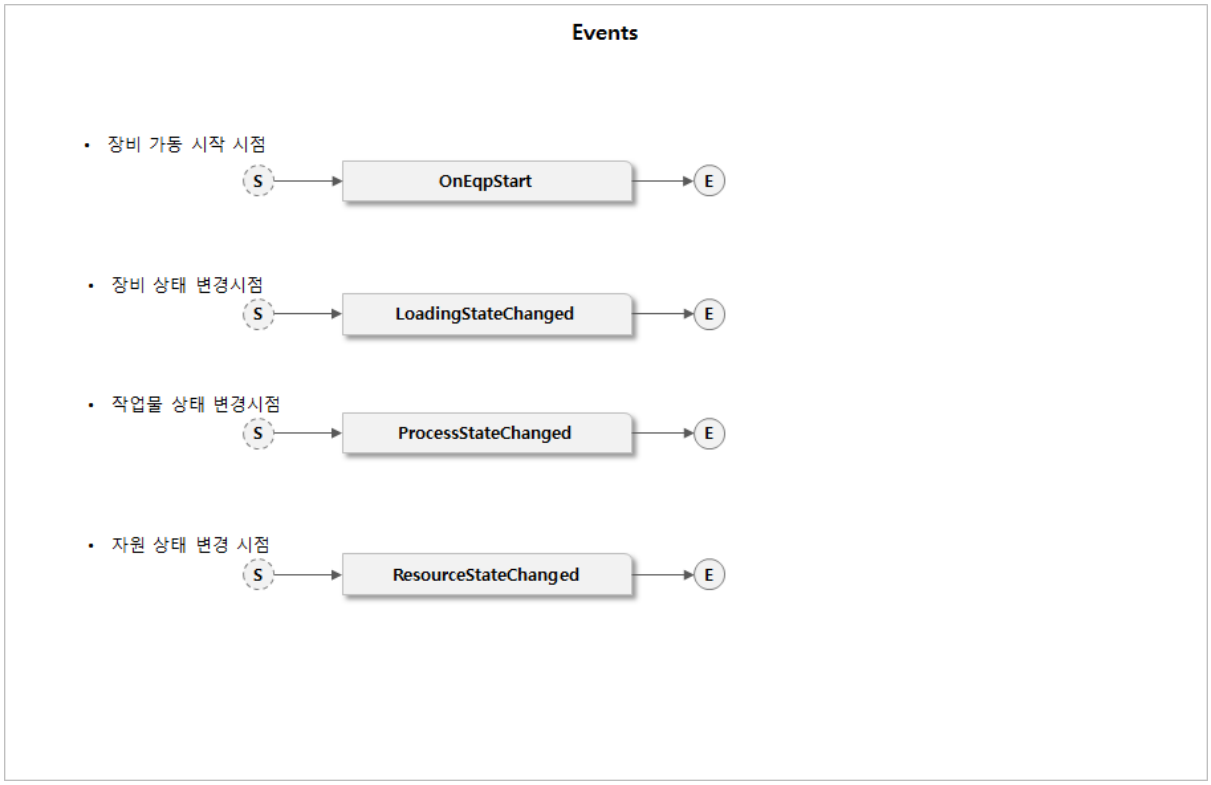
```
public virtual bool IsBatchType(AoEquipment aeqp, ref bool handled, bool prevReturnV
alue)
{
    if (aeqp.Target.SimType == SimEqpType.LotBatch ||
        aeqp.Target.SimType == SimEqpType.BatchInline)
        return true;
```

```
    else
        return false;
}
```

4. ***CanEnter*** : 호출되는 시점에 장비에 추가 로딩이 가능한지 여부를 판별합니다. 기본적으로 장비의 Process 상태, 장비의 Down 상태, Loader 의 Blocking 여부 등은 체크하나, 그 외의 조건에 따라 장비로딩을 제한하고자 할 때 본 Action 을 구현합니다.
5. ***GetChamberIDs*** : 장비의 SimType 이 ParallelChamber 형태인 경우에 해당 장비의 각 Chamber 별 ID 를 List 형태로 반환하여 초기화 할 수 있도록 합니다. Parallel Chamber 장비의 경우 반드시 설정이 되어야 합니다.
6. ***UseCustomLoad*** : 장비에 작업물이 로드되는 시점에 작업물의 속성을 변경하는 등의 사용자 정의 Logic 을 처리하기 위해 설정할 수 있습니다. 값이 true 인 경우 **ProcessControl.OnCustomLoad** FEAction 이 실행됩니다. false인 경우 작업물이 특별한 처리 없이 Processing 됩니다.
7. ***UseCustomUnload***: 장비에 작업물이 Unload되는 시점에 작업물 혹은 설비의 속성을 변경하는 등의 사용자 정의 로직 사용 여부를 반환하는 함수입니다. 값이 `true` 인 경우 **ProcessControl.OnCustomUnload** FEAction이 실행됩니다. `false` 인 경우 작업물이 특별한 처리 없이 Track Out됩니다.

Eqp Events

Process, Setup, Down 의 상황을 제어하기 위한 부분 외에 장비에서 발생하는 Event 에 대한 사용자 정의 Logic 구현을 위한 함수입니다. 해당 Event 가 발생하는 시점에 호출됩니다.



1. ***OnEqpStart*** : 장비는 Simulation 초기화 시점에 생성되며, Simulation 이 시작되는 시점에 개별 장비별로 본 Action 이 호출됩니다. 이 때 추가적이며, 동적인 장비속성을 초기화 할 수 있습니다.
2. ***LoadingStateChanged*** : 장비의 LoadingState 가 변경되는 시점에 이전/현재 상태를 기준으로 처리할 Logic 구현합니다.

> **Note**
> 장비의 상태코드 (LoadingStates) 는 다음과 같은 유형이 있습니다.

- **SETUP** : 장비 SETUP 상태
- **BUSY** : 작업물 가공 상태, Inline 형태의 장비에서는 투입 Capacity 가 여유있는 경우 지속 투입이 발생하는 시점에만 BUSY 상태가 됩니다.
- **IDLERUN** : INLINE 형태의 장비에서 현재 작업물을 가공중이기는 하나, 추가적인 투입이 가능한데 투입이 발생하지 않는 경우 IDLERUN으로 분류합니다.
- **IDLE** : 장비가 UP 상태이나 비가동 중인 상태입니다.
- **PM** : PM에 의한 장비 DOWN 상태입니다.
- **DOWN** : FAILURE에 의한 장비 DOWN 상태입니다.
- **WAIT_SETUP** : 장비에서 SETUP 대기로 비가동 중인 상태입니다.

3. ***ProcessStateChanged*** : 작업 중인 작업물의 작업상태가 변경되는 시점에 처리할 수 있는 사용자 Logic 을 구현합니다.

> 💡 **Note**
> 작업물의 상태는 다음과 같은 순서로 변이가 발생합니다.

- **FL(First Loading)** : 작업물을 구성하는 첫 Unit 이 Processing 시작
- **LL(Last Loading)** : 작업물을 구성하는 마지막 Unit 이 Processing 시작
- **FU(First Unloading)** : 작업물을 구성하는 첫 Unit 이 Processing 종료
- **LU(Last Unloading)** : 작업물을 구성하는 마지막 Unit 이 Processing 종료

4. ***ResourceStateChanged*** : 장비(Resource)의 상태가 변경되는 시점에 처리할 수 있는 사용자 Logic 을 구현합니다.

> 💡 **Note**
> 장비 상태유형은 다음과 같습니다.

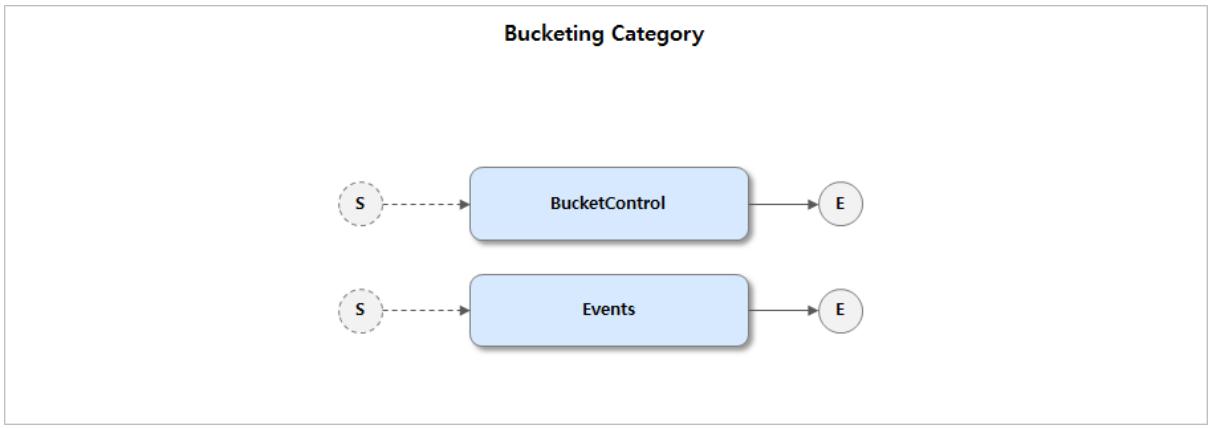
- **UP** : 가동 상태입니다.
- **DOWN** : 다운 상태입니다.

Bucketing

F/W Loading 기반의 Simulation 을 할 때, 장비 로딩을 하지 않고 작업물의 공정을 처리하기 위한 방법으로 Bucketing 을 사용합니다. Bucketing 을 사용하는 경우는 아래의 경우로 요약될 수 있습니다.

1. 특정 공정의 진행 시 장비 정보가 명확하지 않거나 수작업 처리 공정으로 Capacity 만 알 수 있는 경우
2. 공정 In/Out 계획을 위해 물량 Capacity 단위의 다양한 제약이 적용되어 로딩 방식으로 제약을 만족하는 것이 오히려 계획의 정확성을 떨어뜨리는 경우
3. 장비 로딩을 해야 하는 경우이지만 기준정보 오류, 미흡 등의 사유로 특정 공정의 진행가능 장비정보가 누락되는 경우

상기 시점에 Bucketing Logic 을 구현하기 위해 Library 에서 제공하는 Control 은 다음의 두 가지 Component 로 구성됩니다.



BucketControl

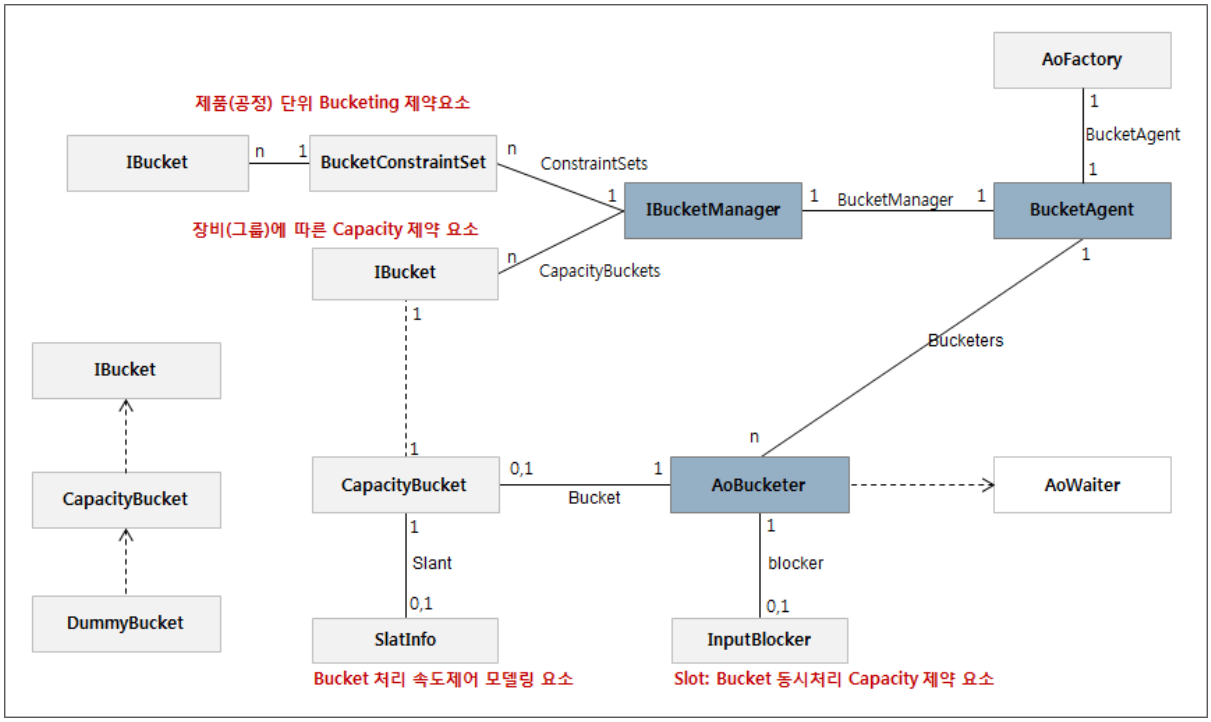
Bucket 에 작업물이 들어오는 시점부터 작업을 마치고 Bucket 을 나가는 시점까지 Bucketing Logic 중 Customizing 가능한 함수를 제공합니다.

BucketEvents

Bucketing Logic 에서 발생하는 Event 에 사용자 Logic 을 추가할 수 있습니다.

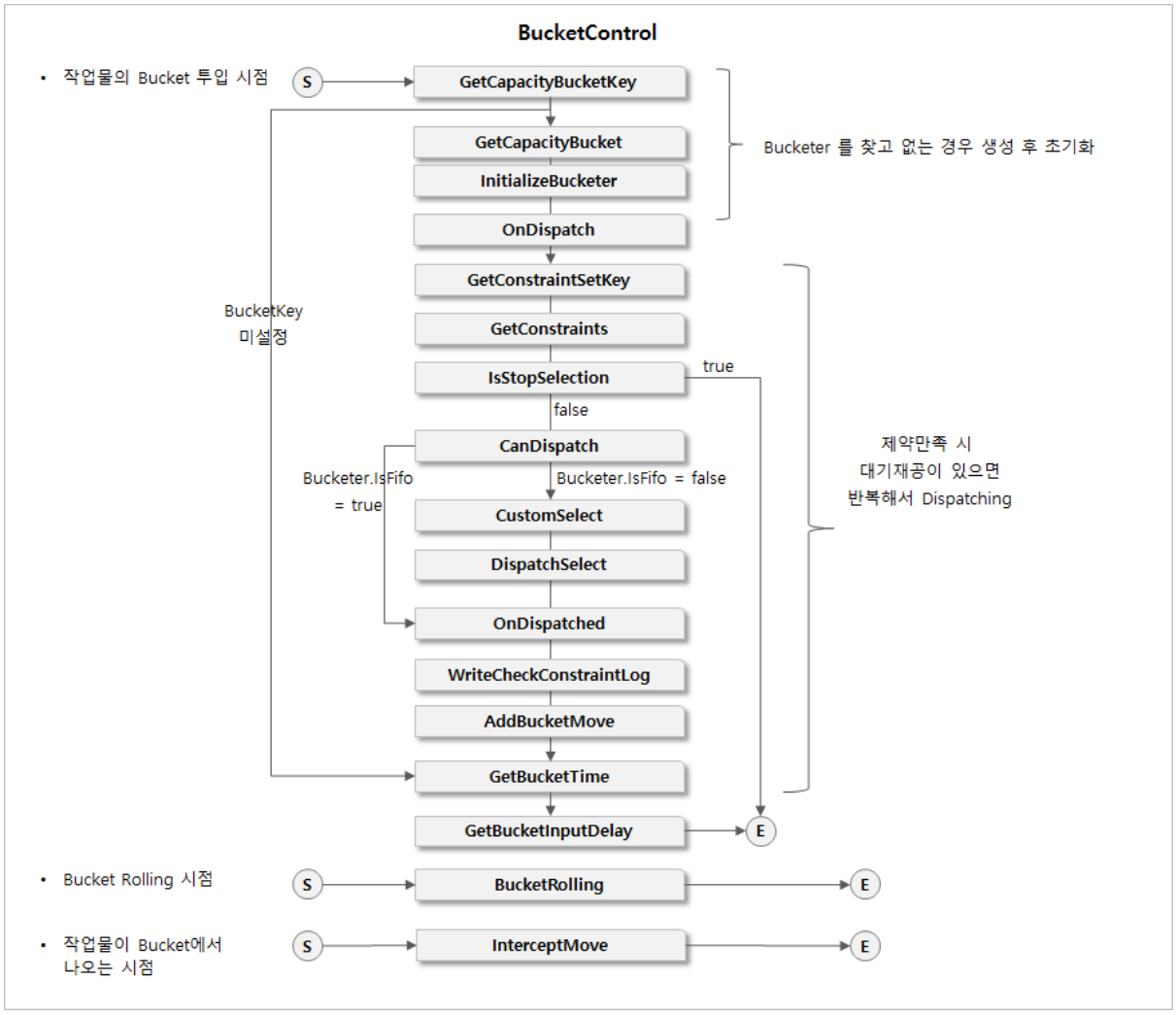
그 외에 기본적인 Bucket 생성 및 초기화 Logic 은 Factory 의 **BucketInit Component** 를 통해 구현할 수 있습니다.

Bucketing 을 위한 SeePlan Library Class 구조



BucketControl 컨트롤

BucketControl 은 Bucketing 대상 작업물이 Bucketing 을 시작하는 시점부터 끝나는 시점까지의 Logic 을 제어합니다. Bucketing 대상 Resource 의 Capacity 가 주어진 경우 기본적으로 해당 Capacity Bucket 을 찾고 Bucket 의 속성에 따라 작업물 및 작업시간 등을 제어하는 함수로 구성되어 있습니다. 아래는 BucketControl 의 제어 흐름을 표시합니다.



1. ***GetCapacityBucketKey*** : 작업물, 공정을 기준으로 대상 Bucket 을 찾는 Key 를 반환하는 FEAction 입니다. 대상 사이트의 Bucket 관리 방식에 따라 Rule 이 정의되고 정의된 Rule 을 구현합니다. Bucketing 수행 시 본 함수를 구현하지 않아 Key 가 설정되지 않은 작업물은 Dummy 처리됩니다. Dummy 처리 시에는 상기 Flow 에서처럼 GetBucketTime 함수부터 실행을 하게 됩니다. 즉, Dummy 로 진행하는 경우에는 제약을 반영하지 못합니다. BucketKey 가 동일한 작업물은 동일한 Bucketer 를 통해 Bucketing 됩니다.

   구현 예시

```
public object GETCAPACITYBUCKETKEY(IHandlingBatch hb, ref bool handled, object prevR
eturnValue)
{
    SimpleMfgSemiconLot lot = SimHelper.GetLot(hb);

    // StepID 문자열 규칙에 따라 Bucketing 을 나누어서 진행하도록 설정하는 예제입니다.
    if (lot.CurrentStep.StepID.StartsWith("S50") || lot.CurrentStep.StepID.StartsWit
h("S70"))
        return lot.CurrentStep.StepID.Substring(0, 3);
```

```
    return null;
}
```

2. ***GetCapacityBucket*** : 상기 BucketKey 에 상응하는 Capacity Bucket 을 반환합니다. 만일 BucketInit 부분에서 해당 Bucket 이 생성되지 않은 경우 해당 Key 에 사용될 Capacity Bucket 을 생성하여 반환합니다. 상기 함수 (GetCapacityBucketKey) 에서 Key 를 설정하고 해당 Key 에 대한 Bucket 을 생성해주지 않은 경우에도 대상 작업물은 Dummy 로 처리됩니다. 반환하는 Bucket 의 속성 중 Enable 속성을 false 로 처리한 경우에도 Dummy 처리됩니다.

   구현 예시

```
public CapacityBucket GETCAPACITYBUCKET(BucketManager bm, object key, ref bool handl
ed, CapacityBucket prevReturnValue)
{
    CapacityBucket cb = new CapacityBucket();

    // S50, S70 모두 동일한 Capacity 를 갖는 Bucketer로 설정합니다.
    switch (key.ToString())
    {
        case "S50":
        case "S70":
            cb.CapaQty = 1000;
            cb.Key = key.ToString();
            cb.Enabled = true;  // Bucketing 을 수행하도록 합니다. 설정하지 않으면 Bucket 에
서 Dispatching 을 하지 않고 Dummy 처럼 동작합니다.
            break;
    }

    return cb;
}
// 만일 Dummy 이지만 Key 따라 다른 DummyBucket 을 사용하도록 하고 싶은 경우
// CapacityBucket 을 생성하지 말고 CapacityBucket 의 Creat
```

3. ***InitializeBucketer*** : Bucket 의 초기화를 수행합니다. Bucketer 에서 Dispatching 수행할 때 FIFO 방식이 아닌 사용자 정의 방식의 작업물 우선순위를 판별하고자 하는 경우 본 함수에서 Bucketer 의 속성을 설정할 수 있습니다.

4. ***OnDispatch*** : Bucketing 을 위한 대기 재공에 대한 Dispatching 을 시작하는 시점에 실행되는 함수입니다.

5. ***GetConstraintSetKey*** : Bucket 의 생산 Capacity 외에 Bucket 을 통해 가공되는 제품(공정)별 제약을 처리하기 위한 Bucket Key 를 반환합니다. 일반적으로 Bucket 을 통해 작업되는 공정의 경우 해당 공정의 제품별로 일별, Shift 별 처리할 수 있는 생산량 한계(Upper Limit) 가 정해져 있는 경우가 많으며, 이러한 제약은 동일 제품에 대해 복수 개를 동시에 만족시켜야 하는 경우도 발생합니다. 이러한 Constraint Bucket Set 을 정의하는 Key 를 반환하도록 Logic 을 구현합니다.

6. ***GetConstraints*** : Constraint Bucket Key 에 상응하는 Bucket Set 을 반환합니다. 아래 예시와 같이 Constraint Bucket 을 만들 때에도 CapacityBucket 클래스를 사용합니다. Constraint 에서 사용되는 Bucket 의 Capacity 가 변동되지 않은 경우에는 **Factory > BucketInit > InitializeBuckets** 에서 ConstraintBucket 을 등록해두고 본 함수에서 등록된 ConstraintBucket 을 찾아 반환합니다. 만일 Bucket 의 Capacity 가 Shift 및 일정 주기 단위로 변경되어 사전에 등록할 수 없는 경우에는 신규로 Bucket 을 만들어 반환합니다. 이 경우에는 자동으로 Rolling 되지 않고 사용자가 별도의 Rolling 처리를 해주어야 합니다. 아래 구현 예시는 후자에 해당합니다.

   구현 예시

```
public IEnumerable<IBucket> GETCONSTRAINTS(IBucketManager bm, IHandlingBatch hb, Dat
eTime now, ref bool handled, IEnumerable<IBucket> prevReturnValue)
{
    string key = GetConstraintKey(hb);
    List<IBucket> consts = new List<IBucket>();
    if (string.IsNullOrEmpty(key))
        return consts;

    CapacityBucket cb = new CapacityBucket();
    cb.Key = key;
    cb.CapaQty = 400;
    consts.Add(cb);

    return consts;
}
```

7. ***IsStopSelection*** : Bucket 에서 Dispatching 을 중단할지 여부를 판별합니다. 대기 중인 Lot 이 많이 남은 상태라도 IsStopSelection = true 인 경우 Dispatching 을 중단합니다.
8. ***CanDispatch*** : Bucket 의 대기 중인 작업물 중 특정 작업물의 Dispatching 이 가능한지 판별합니다. 본 함수의 결과값이 true 인 경우만 해당 작업물의 선택이 가능해집니다. false 를 반환하는 경우에는 해당 작업물은 Bucketing 에서 제외됩니다. 다만, IsStopSelection 과 같이 Dispatching 이 중단되지는 않습니다.
9. ***CustomSelect*** : Bucket 의 대기 Wip 이 있는 경우 작업물에 대한 우선순위를 평가하여 순서대로 작업물을 처리하기 위한 Logic 을 구현할 수 있습니다. Bucketer 의 기본 Dispatching 방식은 FIFO 이며, FIFO Logic 을 사용하지 않고 사용자 정의형으로 Dispatching 을 하기 위해서는 InitializeBucketer 등의 초기화 Action 을 통해 Bucketer 의 FIFO 속성을 설정해주어야 합니다. IsFifo = false 인 경우에만 본 함수를 호출하여 작업물을 선택합니다. 본 함수는 두 개의 작업물의 우선순위를 비교하여 우선순위가 높은 작업물을 반환하여 Logic 을 구현하도록 되어있습니다. 대기 작업물에 대해 본 함수를 사용하여 최우선 작업물을 선택하도록 Logic 이 구성되어 있습니다.

10. ***DispatchSelect*** : Bucketer 에 WeightPreset 이 설정된 경우에는 상기 CustomSelect 로직이 적용되지 않고 Preset 를 사용한 Weight 방식에 따라 Lot 을 평가합니다. 이때, 평가를 마친 정렬된 Lot 중 최종 Lot 을 선택하는 FEAction 입니다.

11. ***OnDispatched*** : Bucketer 에서 Dispatching 이 끝나고 작업물을 선택한 시점에 실행되는 함수입니다. 검증이나 Debugging 등에 사용되며, 기타 해당 시점의 정보를 기록하는 용도로 사용될 수 있습니다.

12. ***WriteCheckConstraintLog*** : Constraint Bucket 의 Check 결과를 기록하기 위한 함수입니다. Constraint Bucket 의 만족여부를 판단한 직후에 호출됩니다. WriteCheckConstraintLog 함수의 Parameter 중 violate 가 null 이면 제약에 걸린 Bucket 이 없이 작업물의 투입이 잘 이루어졌다는 뜻이며, violate 가 null 이 아니면 해당 Bucket 의 제약때문에 작업물의 투입이 안 되었다는 뜻입니다.

13. ***AddBucketMove*** : 작업물이 Bucketer 의 Dispatching 으로 선택된 경우 해당 작업물의 수량만큼 Bucket 의 Capacity 를 차감 (실적 업데이트) 해주어야 합니다. 대상 Bucket 의 Capacity 설정 방식에 따라 처리방식이 바뀔 수 있습니다. 일반적으로 Capacity 는 생산수량으로 설정하나 시간단위의 설정도 가능하며, 이때 본 함수를 통해 대상 작업물의 Bucketing Capacity 를 차감할 양에 대한 계산결과를 반환합니다.

    Default Logic Example

    ```
    public void ADD_BUCKET_MOVE_DEF(CapacityBucket bucket, IHandlingBatch hb, ref bool h
    andled)
    {
        int qty = 0;
        hb.Apply((x, __) => qty+=x.UnitQty);
        bucket.MoveQty += qty;
    }
    ```

14. ***GetBucketTime*** : Bucketing 시 Processing Time 을 산정하여 반환하는 FEAction 입니다. Bucket 의 동시진행 Capacity 및 Volume Capacity 가 지정된 경우에는 장비 로딩 방식과 유사함에 따라 Processing Time 을 지정할 수 있으나, 그렇지 않은 경우 일반적으로 공정의 TAT 를 사용하게 됩니다. 이는 대상 시스템의 운영방식에 따라 변경될 수 있어 사용자가 해당 FEAction 을 통해 사용자 정의 로직을 구현해야 합니다.

15. ***GetBucketInputDelay*** : Bucket 의 Capacity 를 초과하여 추가적인 Input 을 제한하고 있는 상황일 때 현재 Bucket 에 추가 투입이 가능한 시점을 반환하도록 로직을 구현합니다.

    Default Logic Example

    ```
    public Time GET_BUCKET_INPUT_DELAY_DEF(IHandlingBatch hb, AoBucketer bucketer, ref b
    ool handled, Time returnValue)
    {
    ```

```
    var move = bucketer.Bucket;
    if (bucketer.RollingHours <= 0
        || move.CapaQty == 0
        || move.CapaQty > int.MaxValue / 2
        || move.SlotCount == 0)
    {
        return Time.Zero;
    }
    double delay = ((double)bucketer.RollingHours / (double)move.CapaQty) * (double)
move.SlotCount;

    return Time.FromHours(delay);
}
```

16. ***BucketRolling*** : BucketRolling 은 시스템 내부적으로 **[Factory > BucketInit]** 의 속성으로 설정한 **InnerRollingMinutes** 단위로 발생합니다. 기본적으로 시스템에서 제공하는 Definition 에 상기 Logic 이 구현되어 있습니다. 다만, 사용자 조직에서 Rolling 시 별도의 Logic 을 구현하려면 본 함수의 기본 Definition 을 제거하고 추가하여 사용할 수 있습니다. Bucket 에 Slant 를 설정한 경우에는 Rolling 주기별로 MoveQty 가 Reset 되며, **BucketRollingHour** 단위로 CumulativeQty (누적실적) 이 초기화 되어 Capacity 를 설정한 만큼 추가적으로 작업이 가능한 상태가 됩니다. 기본 Definition 을 사용하지 않는 경우 사용자가 유사한 방식으로 처리해주어야 합니다.

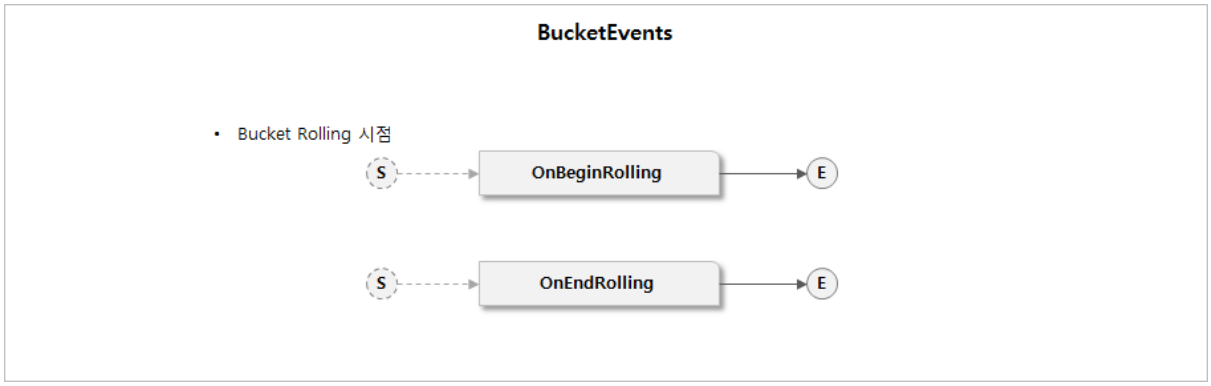
    Default Logic Example

```
public void BUCKET_ROLLING_DEF(CapacityBucket cb, DateTime now, bool atBoundary, boo
l atDayChanged, ref bool handled)
{
    if (atBoundary)
    {
        //cb.PreviousQty = 0;
        cb.CumulatedQty = 0;
        cb.MoveQty = 0;
    }
    else
    {
        cb.PreviousQty = cb.MoveQty;
        cb.CumulatedQty += cb.MoveQty;
        cb.MoveQty = 0;
    }
}
```

17. ***InterceptMove***: 작업물이 장비에서 최종 Track Out 되어 Transfer 로 넘어가기 전에 작업물의 Split/Merge 등의 작업을 수행할 수 있는 FEAction 입니다.

BucketEvents

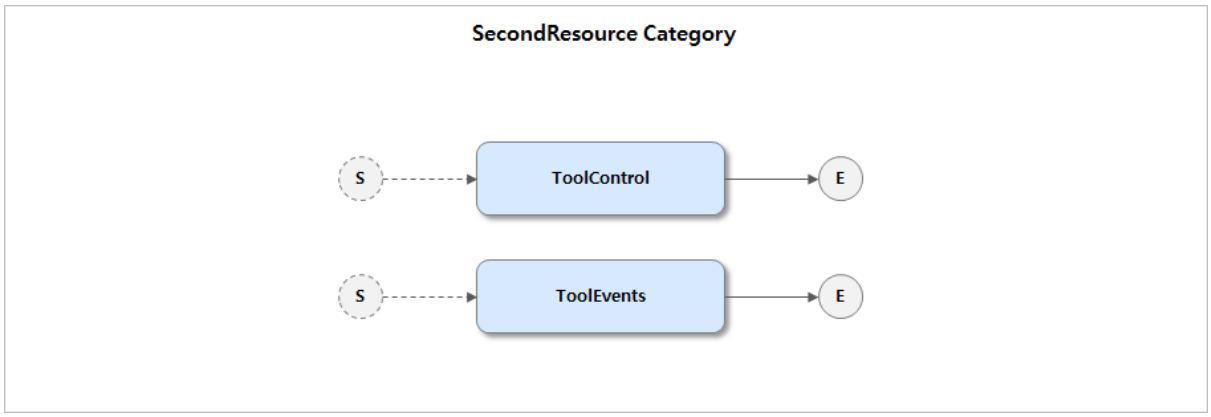
Bucketing Logic 을 담당하는 객체에서 발생할 수 있는 사용자 정의가능 Event 의 집합입니다.



1. ***OnBeginRolling*** : Bucket Rolling 시작 시점에 발생하는 Event 입니다. 본 Event 직후 모든 Bucket (Capacity, Contraint) 에 대한 Rolling 처리가 진행됩니다. Rolling 시 발생되는 Event 처리 함수는 **Bucketing>BucketControl>BucketRolling** 입니다.
2. ***OnEndRolling*** : Bucket Rolling 종료 시점에 발생하는 Event 입니다.

SecondResource

공정 진행시 장비에 장착되어야 하는 Tool, Setup을 위해 필요한 인력과 치공구, Batch 공정을 위해 작업물을 장착하는 Rack 등 가공 시 필요한 2차적인 자원들을 관리하고 제어하기 위한 FEComponet 입니다.



Second Resource 의 경우 SetupCrew 와 같은 특화된 형태의 Resource 를 제외하고 크게 작업물에 장착되는 형태와 장비에 장착되는 두 가지 형태의 Resource 로 구분할 수 있으며, 각

유형별로 실제 처리되는 방식이 약간 다르게 됩니다.

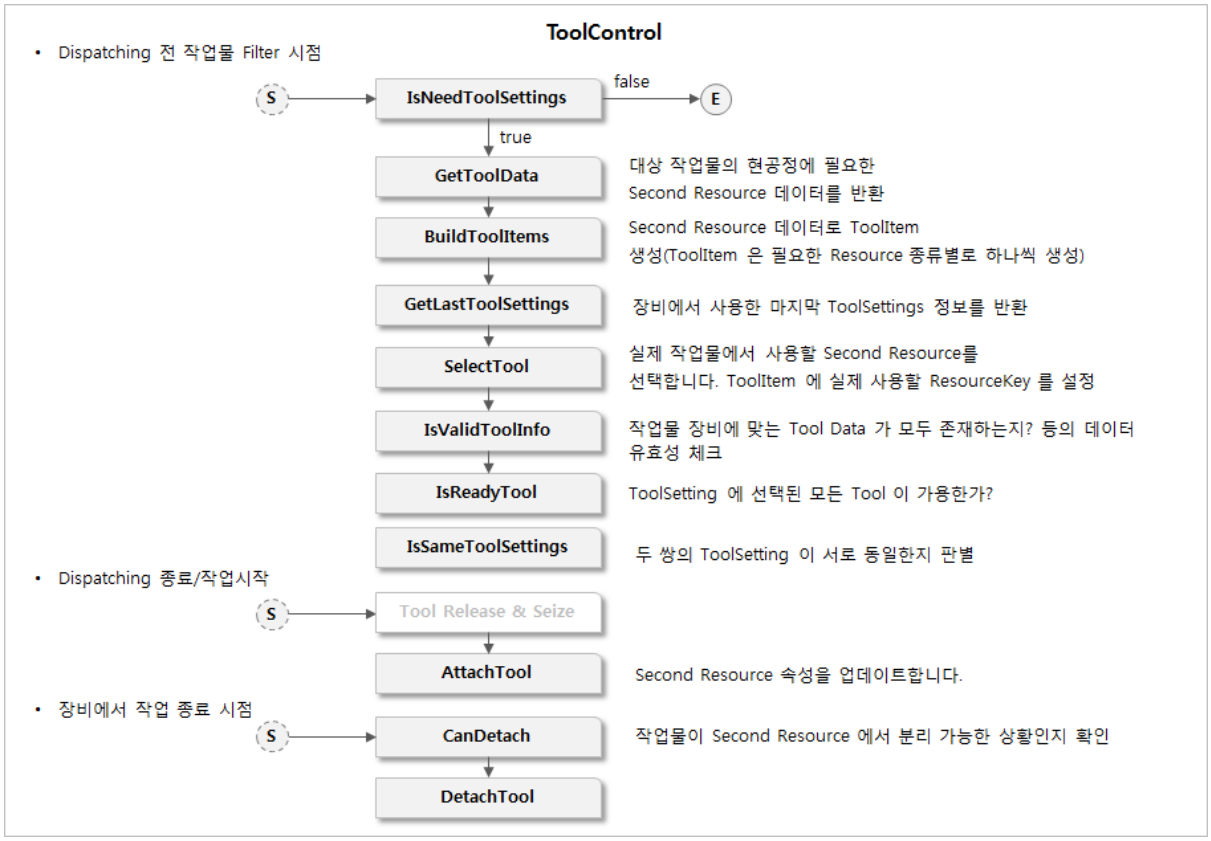
ToolControl
Second Resource 의 선택, 유효성 검사, 점유, 해제 등 기본적인 Resource 사용 제어 전체를 처리하는 함수의 집합입니다.

ToolEvents
Second Resource 에 대한 Simulation Model 상에서 발생하는 Event 시점에 호출되는 Event 함수 집합으로 사용자 정의 Logic 을 구현할 수 있습니다.

ToolControl 컨트롤

Second Resource 의 선택, 유효성 검사, 점유, 해제 등 기본적인 Resource 사용 제어 전체를 처리하는 함수의 집합입니다. 작업물을 특정 공정을 처리하기 위해 필요한 Second Resource 는 n 개가가 될 수 있으며, SEEPLAN Library 에서 이에 대한 클래스 Model 은 ToolSettings 입니다. ToolSettings 는 여러 Second Resource 의 제어를 위해 Second Resource 에 Mapping 되는 복수의 ToolItem 을 포함할 수 있습니다.

ToolSettings 는 Dispatching 직전에 작업물 Filtering 되는 시점에 설정되며, ToolSet 이 필요한 작업물에 유효한 ToolSet 을 구성하지 못한 경우 작업을 할 수 있도록 제어합니다.



1. ***IsNeedToolSettings*** : 대상 작업물이 현재 공정을 처리하기 위한 Second Resource 집합이 필요한지 여부를 판별합니다.
2. ***GetToolData*** : 대상 작업물의 현재 공정에 필요한 Second Resource 를 선택하기 위한 전체 집합 정보를 반환합니다. 특정 제품의 특정 공정을 처리할 수 있는 장비에 따라 Second Resource가 여러 개가 존재할 수 있으며, 본 함수 호출 시점에는 장비가 지정된 상태임으로 이 때 가용한 Second Resource 집합을 반환합니다. Data 의 반환 형태가 IToolData 로 My Objects 에서 IToolData 를 구현한 클래스를 정의해야합니다.
3. ***BuildToolItems*** : 대상 작업물이 현 공정 처리를 위해 필요한 모든 Second Resource 별 ToolItem 을 생성합니다. 예를 들어 작업물이 장착되어야 하는 Rack 과 장비에 장착되는 Tool 이 필요한 공정인 경우에는 두 개의 ToolItem 을 생성해주어야 합니다. 본 Action 의 구현예시는 아래와 같습니다.

   구현 예시

```
public IEnumerable<ToolItem> BUILDTOOLITEMS(IToolSettings tool, ref bool handled, IE
numerable<ToolItem> prevReturnValue)
{
    List<ToolItem> toolItems = new List<ToolItem>();
    ToolItem rack = new ToolItem("Rack", 1);
    ToolItem tool = new ToolItem("Tool", 1);

    toolItems.Add(rack);
    toolItems.Add(tool) ;
```

```
    return toolItems;
}
```

4. ***GetLastToolSettings*** : 대상 장비에서 마지막에 Setting 된 ToolSettings 정보를 반환합니다. Default Definition 이 있으며, 장비의 CurrentToolSettings 를 반환하도록 코드가 구현되어 있습니다. 장비정보를 수정하여 구현하는 경우 이를 수정할 수 있습니다.

   Default Logic Example

```
public ToolSettings GET_LAST_TOOL_SETTINGS_DEF(AoEquipment aeqp, ref bool handled, T
oolSettings prevReturnValue)
{
    return aeqp.CurrentToolSettings;
}
```

5. ***SelectTool*** : ToolData 에 기반해서 현재 시점에 가용한 Second Resource 를 선정한 후 ToolItem 에 설정할 Second Resource 의 Key 정보를 반환합니다. 예를 들어 ToolData 상으로 현재 시점에 사용할 수 있는 Tool 이 n 개 있을때, 이 중 어떤 Tool 을 사용할 지에 대한 Logic 을 구현할 수 있습니다. Tool 의 가용성 체크를 포함합니다. 본 Action 의 구현예시는 아래와 같습니다.

   구현 예시

```
public object SELECTTOOL(IToolSettings tool, ToolItem item, ILot lot, AoEquipment ae
qp, ToolItem last, bool canAlt, ref bool handled, object prevReturnValue)
{
    ToolArranges loadable = tool.Data as ToolArranges;

    if (loadable.Tools== null || loadable.tools.Count <= 0)
        return null;

    var factory = aeqp.Factory ;

    // 대상 ToolItem 의 SecondResourcePool 명칭으로 Pool 객체를 찾음
    var pool = factory.GetResourcePool(item.ResourceType);

    if (pool == null)
        return null;

    SiteLot slot = lot as SiteLot;
    string resourceKey = string.Empty;
    foreach (SiteTool tool in loadable.Tools)
    {
        // Second resource pool 에서 arrange 상의 tool 을 검색
        var sres = pool.GetResource(tool.ToolID, aeqp);
        if (sres == null)
            continue;

        // 해당 Second Resource 의 가용여부 체크
        if (sres.IsAvailable(aeqp) == false)
```

```
                continue;
        if (last != null && last.ResouceKey.ToString() == tool.ToolID)
        {
            resourceKey = reticle.ReticleID;
            break;
        }
    }
    return resourceKey;  // 대상 ToolItem 인 item.ResourceKey 가 반환값으로 설정됩니다.
}
```

6. ***IsValidToolInfo*** : Tool 선택이 완료된 후 선택된 ToolData 가 작업물, 장비에 대해 유효한 ToolData 인지 필요한 모든 ToolData 가 존재하는지 등 ToolSettings 를 구성하는 Tool Data 의 유효성을 검증합니다. 기본적으로 설정된 ToolItem 의 ResourceKey 가 설정되어 필요한 ToolItem 이 정상적으로 설정되었는지 확인합니다.

   Default Logic Example

```
public bool IS_VALID_TOOL_INFO_DEF(ToolSettings tool, ToolItem current, ref bool han
dled, bool prevReturnValue)
{
    if (current.ResouceKey == null)
        return false;

    return true;
}
```

7. ***IsReadyTool*** : 선택된 Tool (Second Resource) 현재 시점에 사용이 가능한지 판단합니다.

   Default Logic Example

```
public bool IS_READY_TOOL_DEF(ToolSettings tool, ToolItem current, ToolItem last, re
f bool handled, bool prevReturnValue)
{
   if (current.ResouceKey == null)
       return false;
   return true;
}
```

8. ***IsSameToolSettings*** : 두 쌍의 ToolSettings 가 동일한지 여부를 판별합니다.

9. ***AttachTool*** : 제품을 담거나 제품에 부착되는 형태의 Second Resource 를 점유하는 FEAction 입니다. 작업물이 선택된 직후(OnDispatched) 에서 호출됩니다. 이 Action 이 호출되기 직전에 Equipment 가 점유하는 형태의 Second Resouce 의 Release/Seize 처리가 선행됩니다.

   Default Logic Example

```
public void ATTACH_TOOL_DEF(IHandlingBatch hb, ref bool handled)
{
   hb.ToolSettings.Attach();
}
```

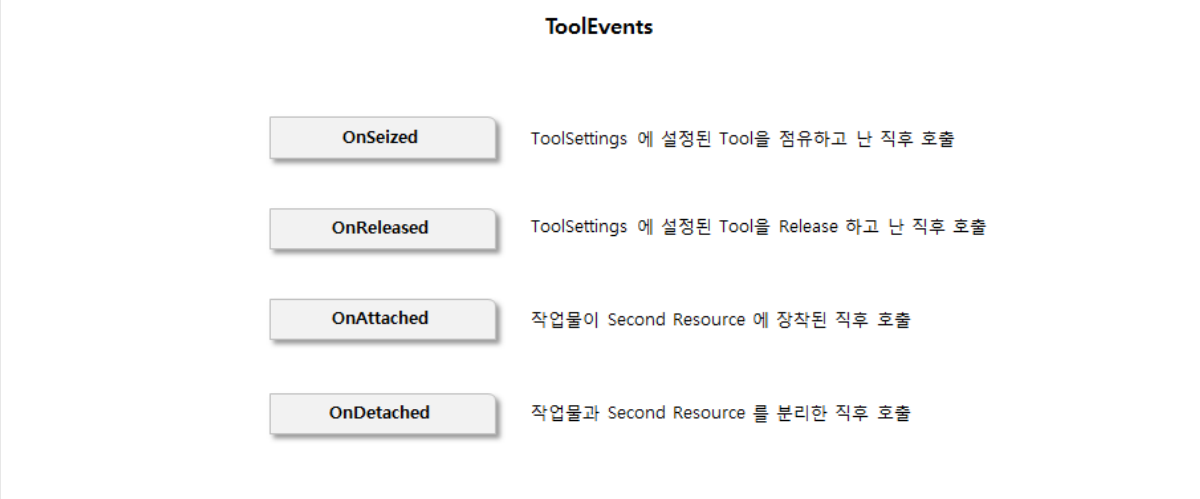
10. ***CanDetach*** : 장비에서 작업이 종료되는 시점에 작업물을 포함하는 Second Resource (e.g. Rack, Cassete 등) 에서 대상 작업물을 분리할 수 있는 상태인지 판별합니다. 장비에서 공정 수행을 위해 여러 작업물을 하나의 Second Resource 에 장착해서 사용하는 경우에는 연속된 공정에서 동일한 Second Resouce 에 장착된 상태로 물류, 공정이 이루어질 수 있습니다. 이 경우 첫 공정이 끝나는 지점에서 대상 작업물을 Second Resource 에서 분리할 수 있는지 여부를 판별하는 Logic 을 구현해야할 필요가 있습니다. 기본값은 true 입니다. 매 공정별로 장착된 Second Resource 를 반환하는 방식입니다. 복수의 단위 작업물 (ILot) 을 하나의 Second Resource 에 장착하여 작업하는 경우에는 작업 시작 전에 작업물을 LotBatch, LotGroup 으로 묶어서 진행해야 합니다.
11. ***DetachTool*** : 제품을 담거나 제품에 부착되는 형태의 Second Resource 의 점유를 해제하는 FEAction 입니다.

    Default Logic Example

    ```
    public void DETACH_TOOL_DEF(IHandlingBatch hb, ref bool handled)
    {
        hb.ToolSettings.Detach();
    }
    ```

Tool Events

Second Resource 의 제어와 관련된 Simulation Model 에서 발생하는 Event 처리 함수의 집합입니다. 해당 Event 처리기의 구현을 통해 추가적인 사용자 정의 Logic 을 개발할 수 있습니다.

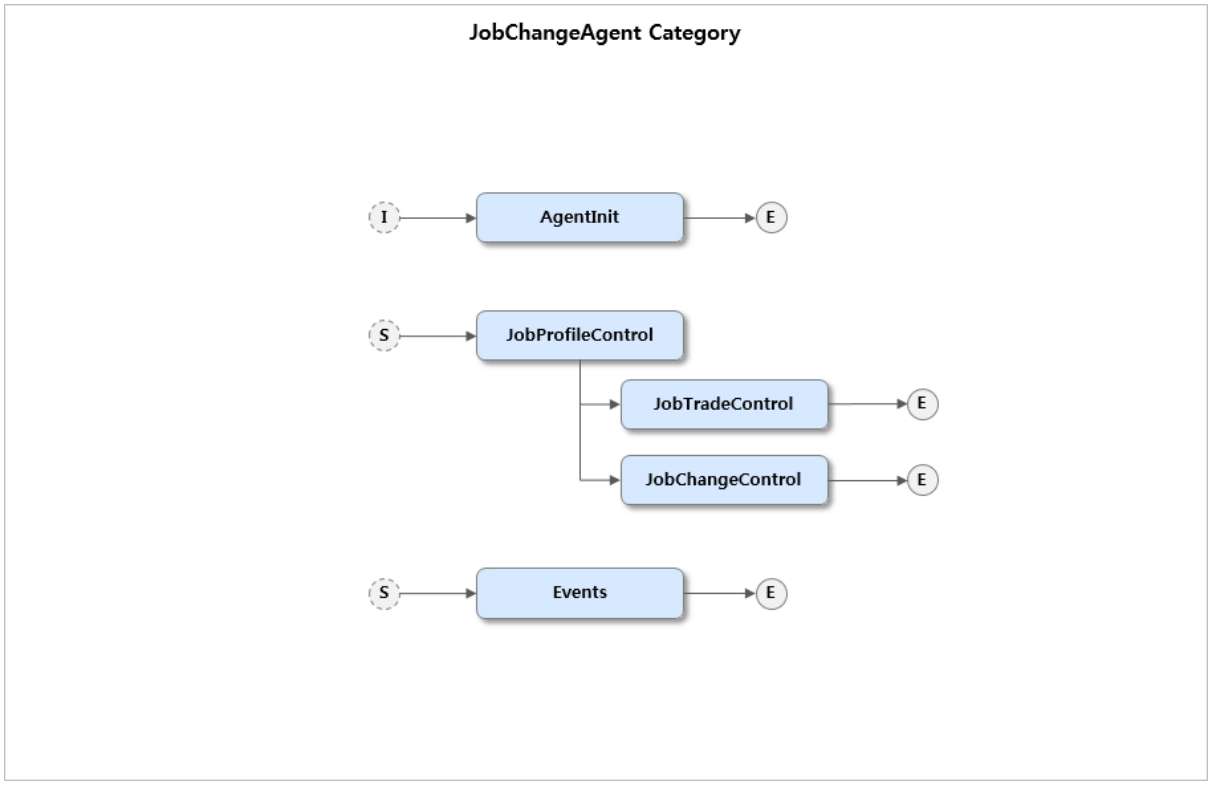


1. ***OnSeized*** : 장비에 장착되는 Second Resource 의 경우 발생하는 Event 로 장비에서 Second Resource 를 점유하고 난 직후 호출되는 Event 입니다.
2. ***OnReleased*** : 장비에 장착되는 Second Resource 의 경우 발생하는 Event 로 장비에서 Second Resource 를 풀어주고 난 직후 호출되는 Event 입니다.
3. ***OnAttached*** : 작업물을 부착하는 Second Resource 의 경우 발생하는 Event 로 작업물을 Second Resource 에 장착 한 후 호출되는 Event 입니다.
4. ***OnDetached*** : 작업물을 부착하는 Second Resource 의 경우 발생하는 Event 로 작업물을 Second Resource 에서 분리 한 후 호출되는 Event 입니다.

Tool (Second Resource) 를 사용 시점, 종료되는 시점에 Second Resource 의 속성값을 업데이트하고, Second Resource 의 사용한계와 같은 제약이 있는 경우 누적 사용량과 사용한계를 비교하여 상태를 PM 으로 변경하거나 사용불가 상태로 변경하는 등의 처리를 수행할 수 있는 FEAction 입니다.

JobChangeAgent

F/W Loading Simulation 은 기본적으로 Pull 방식의 Schedule 을 지원합니다. Dispatching 시점에 대기 재공 중에 우선순위가 가장 높은 재공을 선택하여 진행하도록 Logic 이 구성됩니다. 그러나 장비가 많고 Setup 시간이 오래 걸리는 장비 Group 에 대한 Schedule 시에는 Setup Loss 를 줄이고 장비 효율을 증대하기 위해 재공 외에 해당 장비 Group 을 사용하는 공정별 재공 도착 Pattern 및 제품별 생산목표를 감안하여 장비의 작업교체 의사결정을 수행해야 할 경우가 있습니다. 이러한 의사결정을 지원하는 SEEPLAN 의 Planning 기능이 JobChangeAgent 입니다.



JobChangAgent 는 다음의 다섯 가지 Component 로 구성됩니다.

AgentInit

JobChange 의사결정이 필요한 장비 (공정) Group 을 정의하고 해당 Group 의 유입재공 및 가용 Resource 등의 초기화를 담당하는 Component 입니다.

JobProfileControl

JobChange 의사결정을 수행하기 위한 참고 자료인 재공에 대한 In/Out Profile을 작성하는 Component 입니다. 이 Component 는 JobChange 의사결정 주기마다 호출되며, 한 주기 내에서도 경우에 따라 여러 번 호출될 수 있습니다.

JobChangeControl

JobChange 의사결정을 수행하고 Queue 에 대한 제어를 수행하는 JobChange Main Logic Component 입니다. WorkAgent 의 JobChange 의사 결정 방식 (AgentType) 이 CHANGE 방식인 경우, 이 Component 가 실행됩니다. CHANGE 방식의 JobChange 의사 결정은 Down 대상 공정 (WorkStep) 을 우선 결정하고, Down 된 설비 또는 유휴 설비를 Up 대상 공정 (WorkStep) 에 할당하는 방식을 의미합니다. 이 Component 는 JobChange 의사 결정을 수행할 주기마다 호출됩니다.

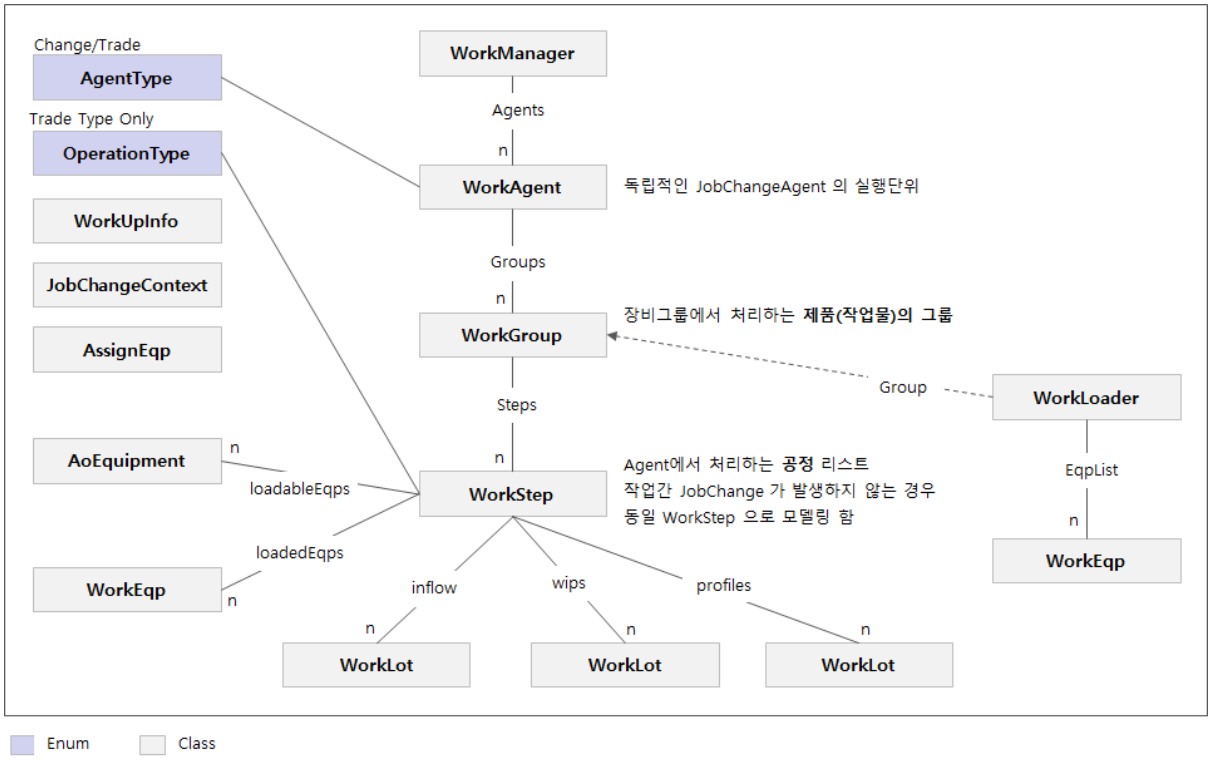
JobTradeControl

JobChange 의사결정을 수행하고 Queue 에 대한 제어를 수행하는 JobChange Main Login Component 입니다. WorkAgent 의 JobChange 의사 결정 방식 (AgentType) 이 TRADE 방식인 경우, 이 Component 가 실행됩니다. TRADE 방식의 JobChange 의사 결정은 Up 대상 공정 (WorkStep) 을 우선 결정하고, 다른 공정 (WorkStep) 에 이미 할당된 설비나 유휴 설비를 Up 대상 공정 (WorkStep) 에 할당하는 방식을 의미합니다. 이 Component는 JobChange 의사 결정을 수행할 주기마다 호출됩니다.

JobChangeEvents

JobChangeAgent Model 의 실행시점에 발생하는 Event 처리 함수 집합입니다. 사용자 정의 Logic 을 해당하는 시점에 추가할 수 있습니다.

JobChangeAgent 를 위한 SeePlan Library Class 구조



클래스 설명

WorkManager 클래스

JobChangeAgent 로직 전체를 관장하는 클래스이며, AoFactory 생성시 자동으로 생성됩니다.

WorkManager

Aa 속성(함수)	Data Type(Return Type)	설명
Duration	Time	Agent Rolling Cycle Time
Initialized	bool	WorkManager가 초기화 되었는지 여부를 가져옵니다. 초기화 되기 전에 추가된 작업물은 임시로 저장되었다가 초기화 시점에 WorkLot 개체로 변경됩니다.
Agents	ICollection<WorkAgent>	WorkManager에 등록된 Agent 리스트
AddLot(IHandlingBatch hb)	void	재공에 Simulation 상에서 춘가되는 시점(Factory.In)에 호출되어 WorkStep에 WorkLot으로 등록함
UpdateLot(IHandlingBatch hb)	void	Simulation 중 작업물의 작업이 시작되는 시점(StartTask), 공정변경시점(Move Next), 공정간 이동 시작 시점(OnTransfer), Dispatching 대기 시작 시점에서 호출되는 함수. 본 함수가 호출될 때 작업물의 현재 상태, 위치에 따라 대상 WorkStep의 WorkLot 별 AvailableTime을 보정해 줄 수 있습니다. **이를 위해 JobProfileControl.Update 함수를 구현해야 합니다.**
RemoveLot(IHandlingBatch hb)	void	재공이 Simulation 종료되는 시점에 호출되어 WorkStep의 재공에서 삭제됨

AgentType (ENUM)

JobChangeAgent 의 로직 유형을 표시하기 위한 Enum type 입니다.

AgentType

Aa 속성(함수)	설명
CHANGE	CHANGE 방식 AGENT 유형
TRADE	TRADE 방식 AGENT 유형

WorkAgent 클래스

JobChange 로직을 독립적으로 실행하는 단위 Agent 클래스입니다. 하나의 시뮬레이션 모델에 완전히 독립적으로 JobChange 로직을 처리할 수 있는 장비그룹이 있는 경우에는 별도의 Agent 를 등록해서 사용할 수 있습니다. 일반적으로 하나의 Agent 로 처리할 수 있습니다.

WorkAgent

Aa 속성(함수)	Data Type (Return Type)	설명
Id	string	Agent ID
AgentType	AgentType(Enum)	AgentType ={"CHANGE, "TRADE"}
Groups	IList<WorkGroup>	Agent에 포함된 WorkGroup 리스트
Interval	Time	Agent의 수행 주기
LastActiveTime	Time	미자막 수행 시각
Factory	AoFactory	AoFactory 객체
LoadableWorkSteps	Dictionary<AoEquipment, List<WorkStep>>	장비별 로딩 가능 WorkStep 정보
Context	JobChangeContext	**AgentType = "TRADE"에서만 사용.** Agent 수행 시점에 전역변수를 저장하기 위한 Context입니다.
AssignedInfos	Dictionary<AoEquipment, AssignEqp>	장비에 해당하는 AssignEqp 정보를 저장하고 있으며, AssignEqp 의 상태와 WorkStep 은 Trade 중 업데이트됩니다. 시뮬레이션 중 특정 AoEquipment 에서 대상 장비의 Agent 수행결과 현재 할당 정보를 얻고 싶은 경우 사용할 수 있습니다.
IsReProfiling	bool	**AgentType = “TRADE” 에서만 사용** 설비할당 변경 후 변경이 일어난 WorkStep이 속하는 WorkGroup의 Profile 재구성 여부입니다. True 로 설정된 경우 장비가 추가로 할당되었거나 작업 중이던 장비를 Release 한 WorkStep 을 포함하는 WorkGroup 의 모든 WorkStep 에 대해 Profile 을 다시 계산합니다. 이 경우 한번 할당된 WorkStep 의 경우에도 다시 Profile 평가 후 UP 판단을 받을 수 있어 1대 이상의 장비를 추가 할당할 수 있습니다.

Aa 속성(함수)	Data Type (Return Type)	설명
IsReleaseDownEqp	bool	**AgentType = "TRADE" 에서만 사용** Down으로 판단된 WorkStep에서 설비 할당 해제 여부입니다. true 로 설정된 경우 "Down"으로 설정된 WorkStep 에서 장비를 Release 합니다. 본 옵션 = true 인 경우에만 JobTradeControl.SelectDownEqp Action 이 활성화됩니다. Trade 가 끝난 후 Down 판정을 받았으나 장비 Down 이 없는 WorkStep 에 대해 해당 함수에서 Down 장비를 결정하도록 합니다. (예제 참조)
AddDummyProfile	bool	Profile 중에 Idle, Setup 등의 유휴시간 발생시 이를 별도의 DummyLot을 생성하여 Profile 을 추가할지 여부를 결정하는 속성, true 인 경우에 유휴시간 발생할 때 마다 Dummy Lot 이 Profile 에 추가됩니다. 이때 Idle 은 기본적으로 추가되며, PM, Setup 등을 고려하려고 하는 경우에는 `JobProfileControl` 의 `GetProfileTimes`, `CreateDummyWorkLot` 함수를 구현해서 세부적인 Loader 의 작업을 모델링 할 수 있습니다. (예제 참조) → CreateDummyWorkLot Action 의 경우 옵션을 사용하는 경우 Default Definition 을 수동으로 추가해주어야 합니다. → 옵션을 사용하는 경우에는 Profile 결과를 처리하는 부분에 항상 WorkLot 의 ProfileType 을 확인하여 알맞은 방식으로 로직을 처리해야 합니다.

WorkGroup 클래스

작업조건이 동일한 제품의 그룹에 대한 클래스입니다. WorkGroup 이 변경되면 JobChange 가 발생함을 의미합니다. JobChangeAgent 에서 Inflow 및 Out Profile 을 평가하기 위한 제품의 그룹입니다.

WorkGroup

Aa 속성(함수)	Data Type(Return Type)	설명
Steps	List<WorkStep>	WorkGroup 에서 JobChange 의사결정 대상이 되는 공정(WorkStep) 리스트
Agent	WorkAgent	WorkGroup이 포함된 WorkAgent

Aa 속성(함수)	Data Type(Return Type)	설명
Factory	AoFactory	AoFactory 객체
Key	object	WorkGroup Key
Data	object	사용자 정의 데이터
Ordered	bool	WorkStep 순서대로 Profile 을 그리는 옵션입니다. 이 옵션이 true 로 설정된 경우에만 `JobProfileControl.CompareProfileStep` 함수를 사용하여 WorkGroup 내 WorkStep 을 소팅한 후 소팅 순서에 따라 할당합니다. 기본값은 false 입니다.
CaclulateProfile()	void	WorkGroup 의 모든 WorkStep 에 대한 In, Out Profile 계산함수. 모든 WorkStep 을 작업중인 장비를 추출하여 해당 장비들의 Profile 을 다시 계산함
CalculateProfile(WorkStep)	void	지정된 WorkStep 에 대해서만 다시 Profile 을 계산하기 위한 함수. 지정된 WorkStep 을 로딩중인 장비에 대해서만 Profile 을 다시 계산함

OperationType (ENUM) - AgentType = "TRADE"에서만 사용

TRADE 방식에서 사용하는 WorkStep 의 OutProfile 평가 후 향후 로직에서 사용하기 위해 WorkStep 에 설정하는 처리 유형에 대한 Enum type 입니다.

OperationType

Aa 속성(함수)	설명
Keep	현재 할당 장비 유지 **[판단 기준]** → 현재 작업중인 장비로 작업하는 경우에도 Target을 달성 할 수 있는 경우 → 장비를 한대 내릴 경우 Target을 달성할 수 없는 경우 → 일정시간 동안 작업이 유지되는 경우
Up	추가 장비 할당 필요 **[판단 기준]** → 생산 Profile 이 일정시간 이상 충분히 확보되는 경우 (유입량 과다) → 현재 할당된 장비에서만 작업할 경우 Target 을 만족하지 못하는 경우. (장비가 충분히 할당하지 못함) → 추가 할당할 경우에도 일정량 이상의 작업유지가 가능한 경우 → 장비 할당이 안되어 있고, 새로 할당하지 못하는 경우 Target 달성 불가한 경우
Down	현재 할당 장비를 할당 해제 필요 **[판단 기준]** → 생산 Profile 이 정해진 일정시간 이하로 계산되는 경우, 충분한 유입이 없는 경우 → 장비를 한대 교체할 경우에도 Target 을 달성할 수 있는 경우 = Target 을 넘기는 작업물이 없는 경우
Hold	판단 유보

WorkStep

Job Change 의사결정이 필요한 장비그룹에서 처리하는 공정입니다. WorkGroup 별로 WorkStep 생성됩니다. 동일한 공정이더라도 WorkGroup 이 다른 경우 다른 WorkStep을 생성됩니다. WorkGroup이 동일하더라도 WorkStep 이 다른 경우 JobChange 가 발생하며, 최종적으로 Profile 을 생성하여 장비의 추가할당, 작업중인 장비의 작업교체를 결정하는 단위를 나타냅니다.

WorkStep

Aa 속성(함수)	Data Type(Return Type)	설명
Group	WorkGroup	WorkStep이 포함된 WorkGroup
Agent	WorkAgent	WorkStep이 포함된 WorkAgent
Factory	AoFactory	AoFactory 객체
Key	object	WorkStep의 Key 정보 (Identifier)
Steps	IList<Step>	해당 WorkStep 과 Mapping 되는 실제 Route 상의 Step 정보 집합. 통상은 하나의 Step 이 Mapping 되지만 만일 제품 그룹에 포함된 여러 제품의 다른 Step 이 대상 장비에서 작업조건이 같아 Setup 이 발생하지 않는 경우 Workstep : step = 1: n 관계가 가능함
LoadableEqps	ISet<AoEquipment>	대상 WorkGroup, WorkStep 을 로딩가능한 장비 리스트
LoadedEqps	ICollection<WorkEqp>	현재 대상 Group, WorkStep 을 작업중인 장비 리스트
LoadedDictionary	IDictionary<string, WorkEqp>	현재 대상 Group, WorkStep 을 작업중인 WorkEqp Dictionary, Key = 장비ID
LoadedEqpIDs	List<string>	현재 대상 WorkStep 을 로딩중인 장비의 ID 리스트
LoadedEqpCount	Int	현재 WorkStep 을 작업중인 장비의 수

Aa 속성(함수)	Data Type(Return Type)	설명
InFlows	SearchQueue<WorkLot>	해당 WorkStep 의 현재공 + 유입대상 재공, 등록된 시점부터 실제 작업물이 해당 공정에서 Schedule 되기 전까지 임의로 삭제되지 않음
Wips	List<WorkLot>	Profile 을 그리는 중간에 계속 변경되는 재공정보. 특정 장비의 Profile 을 계산하기위한 WorkLoader 에서 사용하는 재공정보. Profile 을 그리기 전에 Reset 후 Inflow 의 재공으로 초기화 된 후 Loader 가 작업물을 로딩 하면서 계속 업데이트 됨
Profiles	MultiDictionary<IHandlingBatch, WorkLot>	WorkLoader 에서 로딩된 작업물의 정보. Profile 계산 시 초기화됩니다.
OperationType	OperationType(Enum)	WorkStep 처리 유형. AgentType = "TRADE" 방식에서만 사용
Reason	object	OperationType 판단 사유. AgentType = "TRADE" 방식에서만 사용
IsDown	bool	WorkStep 을 최종 Down 결정한 경우 현재 WorkStep 의 Down 이 처리되는 시점에 True 로 설정됨. Profile 계산 시마다 초기화됩니다.

Aa 속성(함수)	Data Type(Return Type)	설명
Unassigns	List<ISimEntity>	WorkStep 에서 장비가 Down 되는 경우 해당 장비의 Queue 에서 대기중이던 작업물을 DispatchingAgent 에서 제거하고이 속성에 저장합니다. Workstep에 대해 작업중인 장비가 없는 경우에만 해당 속성을 사용합니다. 만일 해당 WorkStep 이 다시 Up 되는 경우에 Unassigns 에 저장된 lot 들을 Up 장비 Queue 에 추가하는 작업을 라이브러리에서 수행합니다. 따라서 Unassigns 의 Lot 을 사용자가 임의로 처리하는 로직을 만드는 경우 이 관계를 알고 처리하는 것이 필요합니다. (예제 참조)
ProfileAssignWips	ISet<WorkLot>	Profile 계산 시에 반영된 WorkLot 정보입니다. 현 WorkStep의 Out Profile 계산 시마다 초기화됩니다.
FindOverTime(Time decision, Time gap)	WorkLot	Profile 에 포함된 WorkLot 에서 기준으로 param=decision 보다 큰 InTime 을 가지는 WorkLot 을 반환. param=gap 보다 길게 장비의 Idle 이 발생되는 경우에는 마지막 로딩된 작업물을 반환합니다. 현재 오류 있음(수정필요)
AllowedArrivalGap	Time	장비의 Down 이 발생해도 다음 작업물이 도착할 때까지 기다릴 수 있는 한계시간. WorkStep 에 대해 다음 유입되는 재공이 설정된 시간 이후에 도착하면 작업 중인 장비의 작업대상을 교체해야 합니다. → WorkStep 의 FindOver 함수의 주요 argument 로 사용됩니다.

Aa 속성(함수)	Data Type(Return Type)	설명
UpInterval	Time	장비 Up 을 판단하는 유입 재공 양의 작업환산시간. 해당 시간보다 유입재공의 작업 환산시간이 큰 경우 WorkStep 을 처리하는 장비를 늘려주도록 결정합니다. → Change Type 인 경우 `JobChangControl/IsNeedUpStep` 함수의 Default Definition 에서 사용됩니다.
DownInterval	Time	장비 Down 을 판단하는 유입 재공 양의 작업환산시간, 해당 시간보다 유입재공의 작업 환산시간이 적은 경우 WorkStep 을 처리하는 장비를 줄이도록 결정합니다. → Change Type 인 경우 `JobChangControl/IsNeedDownStep` 함수의 Default Definition 에서 사용됩니다.
NewUpInterval	Time	해당 WorkStep 으로 교체하여 작업을 시작한 경우 다른 작업으로 교체하지않고 유지해야 하는 최소 시간입니다. 작업교체 후 최소한 동일 작업을 설정된 시간이상은 할 수 있어야 교체가 가능함을 의미합니다. → Change Type 인 경우 `JobChangeControl/CanUp` 함수의 Default Definition 에서 사용됩니다.
AddLoadedEqp(AoEquipment, bool isReal = true)	void	Loader 를 통해 Profile 를 생성할 때 WorkStep 에 Loading 대상이 되는 장비를 추가합니다. → 장비를 한대 더 붙였을 때 Profile 을 계산해보고자 할 경우 isReal = false 로 장비를 추가한 후 WorkGroup의 CalculateProfile(workstep) 함수를 사용해서 Profile 을 그려볼 수 있습니다. 이후 Profile 을 비교하여 장비 Up/Down 을 판단합니다.

Aa 속성(함수)	Data Type(Return Type)	설명
RemoveLoadedEqp (AoEquipment, bool isReal = true)	void	Loader 를 통해 Profile 를 생성할 때 WorkStep 에 Loading 대상이 되는 장비를 제거합니다.
Data	object	사용자 정의 데이터

WorkLot

JobChangeAgent 의 Inflow(재공), OutProfile 을 구성하기 위한 데이터 클래스입니다. 시뮬레이션의 작업물(Lot) 과 연결정보를 포함합니다.

WorkLot

Aa 속성(함수)	Data Type(Return Type)	설명
Batch	IHandlingBatch	WorkLot 과 연결된 실제 Entity(Batch)
Lot	ILot	WorkLot과 연결된 실제 Entity(Lot)
StepKey	object	WorkLot이 현재 로딩되어야 할 WorkStep의 Key
Step	Step	실제(Simulation의) Lot의 현재 Step 정보
AvailableTime	Time	WorkStep에 도착 가능한 시각
InTime	Time	Profile에 사용된 계획 시작 시간(Loading Start Time)
OutTime	Time	Profile에 사용된 계획 종료 시간(Loading End Time)
InEqp	WorkEqp	Profile에 사용된 로딩 장비
ProfileType	WorkLotType	WorkLotType(Enum) 형으로 WorkLot 의 속성을 정의합니다. = {'Busy', 'Idle', 'Setup', 'PM'}. 'Busy'를 제외한 나머지는 모두 DummyLot으로 유휴시간 정보를 처리하기 위한 가상의 Lot 인 경우 설정됩니다. WorkAgent 의 `AddDummyProfile=true` 인 경우에만 생성됩니다.

WorkUpInfo - AgentType = "Change" 에서만 사용

Change Type WorkAgent 에서 사용되는 JobChange 대상 장비의 변경대상 작업 정보

WorkUpInfo

Aa 속성(함수)	Data Type(Return Type)	설명
Target	AoEquipment	Up 대상 설비 정보
Setups	List<WorkStep>	Setup을 발생시키는 WorkStep 집합

Aa 속성(함수)	Data Type(Return Type)	설명
NoSetups	List<WorkStep>	Setup을 발생시키지 않은 WorkStep 집합
IsEmpty	bool	Setups, NoSetups에 WorkStep이 있는지 여부, true면 비어있다는 뜻

WorkLoader

JobChangeAgent 의 WorkStep 에 대한 OutProfile 을 생성을 관장하는 클래스입니다. Profile 을 생성할 때 WorkGroup 별로 생성하여 작동하고 소멸합니다.

WorkLoader

Aa 속성(함수)	Data Type(Return Type)	설명
EqpList	List<WorkEqp>	Loader에서 로딩 할 대상 장비 리스트
Steps	List<WorkStep>	프로파일을 계산할 WorkStep의 목록을 가져옵니다.
Group	WorkGroup	로딩 대상 WorkGroup (WorkGroup 내의 WorkStep 에 등록된 Wips 를 대상으로 로딩 함)
LockAdvance	bool	WorkLot 이 WorkGroup 내의 다른 WorkStep 으로 이동해서 반복적으로 Profile 을 계산할지 여부를 결정함 기본값은 false 로 설정됨
Load()	void	Loader 에 등록된 장비와 Group 의 재공을 사용하여 실제 가 로딩하여 Profile 을 생성하는 함수

WorkEqp

JobChangeAgent에서 장비의 정보를 참조하기 위해 사용하는 클래스로 주로 WorkLoader 에서 로직을 처리할 때 사용됩니다.

WorkEqp

Aa 속성(함수)	Data Type(Return Type)	설명
Target	AoEquipment	WorkEqp와 연결된 실제 설비
AvailableTime	Time	해당 설비의 다음 가능한 작업시간
IncludeBufferForAvailableTime	bool	설비의 Buffer 안에 있는 작업물을 고려하여 다음 작업 가능한 시간 계산 여부

Aa 속성(함수)	Data Type(Return Type)	설명
Data	object	사용자 정의 데이터
Step	WorkStep	현재 장비가 작업중인 WorkStep 정보
Stop	bool	장비의 추가 로딩을 막을지 여부. true 인 경우 추가적인 loading 불가
IsDone()	bool	로딩중인 WorkStep 의 모든 Wip 이 로딩되었거나, Stop 상태인지 판별

JobChangeContext - AgentType = "TRADE" 방식에서만 사용

Trade 방식의 JobChange 의사결정을 할 때 사용되는 전역변수를 저장하는 클래스입니다.

JobChangeContext

Aa 속성(함수)	Data Type(Return Type)	설명
UpList	List<WorkStep>	장비를 추가로 Assign 할 대상이 되는 WorkStep 집합, Profile계산 후 설정됨
KeepList	List<WorkStep>	현재 상태를 유지할 WorkStep 집합
DownList	List<WorkStep>	장비를 Down 시켜야 할 대상이 되는 WorkStep 집합
HolderList	List<WorkStep>	Up 대상 중 Up 처리가 불가한 WorkStep의 집합
CurrentUpWorkStep	WorkStep	Trade 시점에 선택된 Up 대상 WorkStep
CurrentDownSteps	List<WorkStep>	Trade 시점에 선택된 Down 처리되는 WorkStep 리스트
AllAssignEqps	List<AssignEqp>	Trade 시점에 CurrentUpWorkStep이 로딩 가능한 자이 리스트 (로딩중인 장비는 제외)
FilteredAssignEqps	List<AssignEqp>	Trade 시점에 CurrentUpWorkStep을 추가 할당 가능한 장비 중 조건에 의해 필터링하고 남은 최종 로딩대상 장비 리스트
SelectedAssignsEqps	List<AssignEqp>	Trade 시점에 CurrentUpWorkStep 에 추가로 Assign 하기로 선택된 장비 리스트
AssignedEqps	List<AssignEqp>	Trade 시점에 Agent 실행 중 할당된 장지 리스트
Data	object	사용자 정의 데이터를 저장해두고 JobTradeControl 의 각 로직 수행 시 활용합니다. 로그를 기록하거나 Trade 시점의 글로벌 상태 변수를 정의해서 활용할 수 있습니다.

Aa 속성(함수)	Data Type(Return Type)	설명
AddStep(WorkStep, OperationType, object reason = null)	void	대상 WorkStep을 `OperationType` 유형에 맞게 JobChangeContext에 추가합니다. 이미 대상 WorkStep이 OperationType에 해당하는 목록에 존재하는 경우 이전 WorkStep을 제거한 뒤에 새로운 OperationType에 맞는 목록에 추가합니다. 해당 함수는 기본적으로 `ClassifyOperationType` 함수에서 호출됩니다.

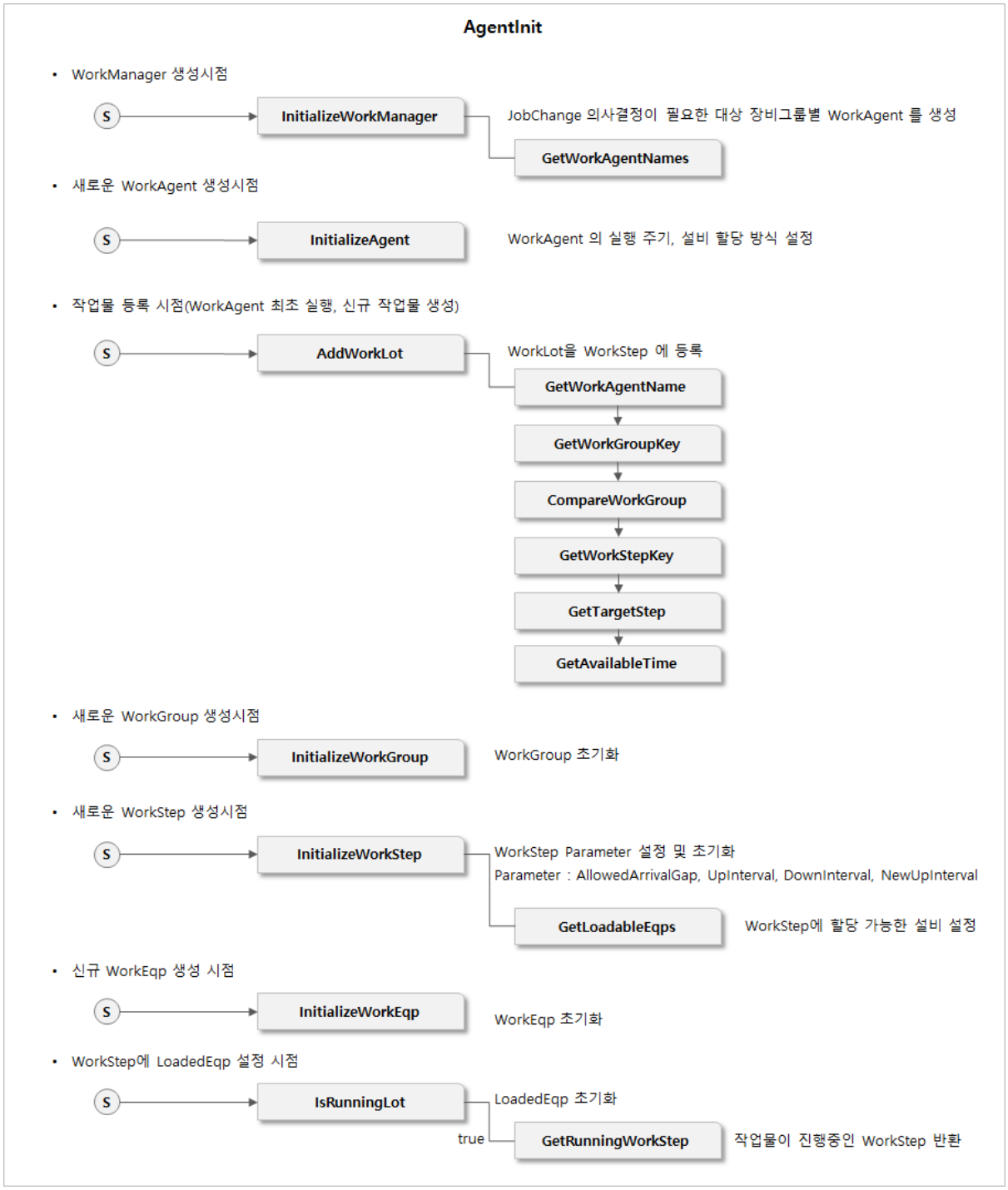
AssignEqp - AgentType = "TRADE" 방식에서만 사용

AssignEqp

Aa 속성(함수)	Data Type(Return Type)	설명
Target	AoEquipment	Mapping된 AoEquipment 객체
WorkStep	WorkStep	설비에 할당된 WorkStep 정보
IsAssigned	bool	Agent 수행 시점에 Assign 되었는지 여부

AgentInit 컨트롤

JobChange 의사결정이 필요한 장비 (공정) Group 을 정의하고 해당 Group 의 유입재공 및 가용 Resource 등의 초기화를 담당하는 Component 입니다.



1. ***InitializeWorkManager*** : JobChangeAgent 전체를 관장하는 WorkManager 의 초기화를 수행합니다. 모델링에 따라 JobChange Agent 를 추가할 대상별로 Agent 를 생성합니다. Agent Key 를 정의하는 작업을 수행합니다.

   Default Logic Example

```
public void INITIALIZE_WORK_MANAGER_DEF(WorkManager wmanager, ref bool handed)
{
    var agentInitControl = ServiceLocator.Resolve<JobChangeInit>();
```

```
    var agents = agentInitControl.GetWorkAgentNames(wmanager);
    if (agents != null)
    {
        foreach (var agent in agents)
        wmanager.Add(agent);
    }
}
```

2. ***GetWorkAgentNames*** : WorkManager 에 추가할 Agent 의 이름을 정의합니다.
3. ***InitializeAgent*** : Agent 의 실행 주기, JobChange 방식을 설정합니다.
4. ***AddWorkLot*** : WorkAgent 에 유입대상 작업물을 등록합니다. 재공 초기화 시점에 각 작업물에 대해 실행되고, 그 이후에는 공장에 새로운 작업물이 생성/투입될 때 (AoFactory.Current.In) 마다 실행됩니다.

Default Logic Example

```
public WorkStep ADD_WORK_LOT_DEF(IHandlingBatch hb, ref bool handled, WorkStep prevR
eturnValue)
{
   var agentInitControl = ServiceLocator.Resolve<JobChangeInit>();

   var wagentName = agentInitControl.GetWorkAgentName(hb);
   if (string.IsNullOrEmpty(wagentName))
       return null;
   var wmanager = AoFactory.Current.JobChangeManger;

   var wagent = wmanager.GetAgent(wagentName);
   if (wagent == null)
       return null;

   var wgroupKey = agentInitControl.GetWorkGroupKey(hb, wagent);
   if (wgroupKey == null)
       return null;

   var wgroup = wagent.GetGroup(wgroupKey);

   var wstepKey = agentInitControl.GetWorkStepKey(hb, wgroup);
   if (wstepKey == null)
       return null;

   var targetStep = agentInitControl.GetTargetStep(hb, wgroup, wstepKey);
   if (targetStep == null)
       return null;

   var wstep = wgroup.GetStep(wstepKey, targetStep);

   var availablaTime = agentInitControl.GetAvailableTime(hb, wstep, targetStep);

   var wlot = new WorkLot(hb, availablaTime, wstepKey, targetStep);
   wstep.AddWip(wlot);

   return wstep;
}
```

5. ***GetWorkAgentName*** : 유입대상 작업물이 속할 Agent 의 이름을 설정합니다.
6. ***GetWorkGroupKey*** : 유입대상 작업물이 속할 WorkGroup 의 Key 를 설정합니다. 설정된 Key 를 갖는 WorkGroup 이 Agent 에 등록되어 있지 않다면 새로운 WorkGroup 을 생성하며, 이 때 해당 WorkGroup 의 초기화를 수행합니다.
7. ***CompareWorkGroup***: WorkGroup간 우선순위를 반영하기 위한 로직을 구현합니다. 이후 ProfileControl에서 WorkGroup 우선순위순으로 Profile를 생성합니다.
8. ***GetWorkStepKey*** : 유입대상 작업물이 속할 WorkStep 의 Key 를 설정합니다. WorkGroup 에 설정된 Key 를 갖는 WorkStep 이 등록되어 있지 않다면 새로운 WorkStep 을 생성하며, 이 때 해당 WorkStep 의 초기화를 수행합니다.
9. ***GetTargetStep*** : 유입대상 작업물의 목표 Step 을 설정합니다. 목표 Step 은 유입대상 작업물이 속한 WorkStep 의 관리대상 Step 중 하나로 등록됩니다.
10. ***GetAvailableTime*** : 유입대상 작업물이 WorkStep 에 도착하는 시간을 설정합니다. 초기화 이후 작업물의 상태가 바뀔 때마다 JobProfileControl 의 Update Action 을 통해 작업물의 도착 시간을 업데이트할 수 있습니다.
11. ***InitializeWorkGroup*** : Agent 에 WorkGroup 이 등록될 때, WorkGroup 의 초기화를 수행합니다.
12. ***InitializeWorkStep*** : WorkGroup 에 WorkStep 이 등록될 때, WorkStep 의 초기화를 수행합니다. WorkStep 의 주요 설정 Parameter 는 아래 노트를 참조하십시오.
    - **AllowArrivalGap** : 장비의 Down 이 발생해도 다음 작업물이 도착할 때까지 기다릴 수 있는 한계시간. WorkStep 에 대해 다음 유입되는 재공이 설정된 시간 이후에 도착하면 작업 중인 장비의 작업대상을 교체해야 합니다.
    - **UpInterval** : 장비 Up 을 판단하는 유입재공량의 작업환산시간, 해당 시간보다 유입재공의 작업 환산시간이 많은 경우 WorkStep 을 처리하는 장비를 늘려주도록 결정합니다.
    - **DownInterval** : 장비 Down 을 판단하는 유입재공량의 작업환산시간, 해당 시간보다 유입재공의 작업 환산시간이 적은 경우 WorkStep 을 처리하는 장비를 줄이도록 결정합니다.
    - **NewUpInterval** : 장비를 한대 더 추가한 경우 Workstep 의 Out Profile 상 지속적으로 작업이 가능한 최소 시간을 설정합니다. 해당 Workstep 에 할당된 모든 장비(추가 장비 포함)가 정해진 시간 이상 연속 작업이 가능해야 새로운 장비를 한대 더 할당 할 수 있다는 개념입니다.

Default Logic Example

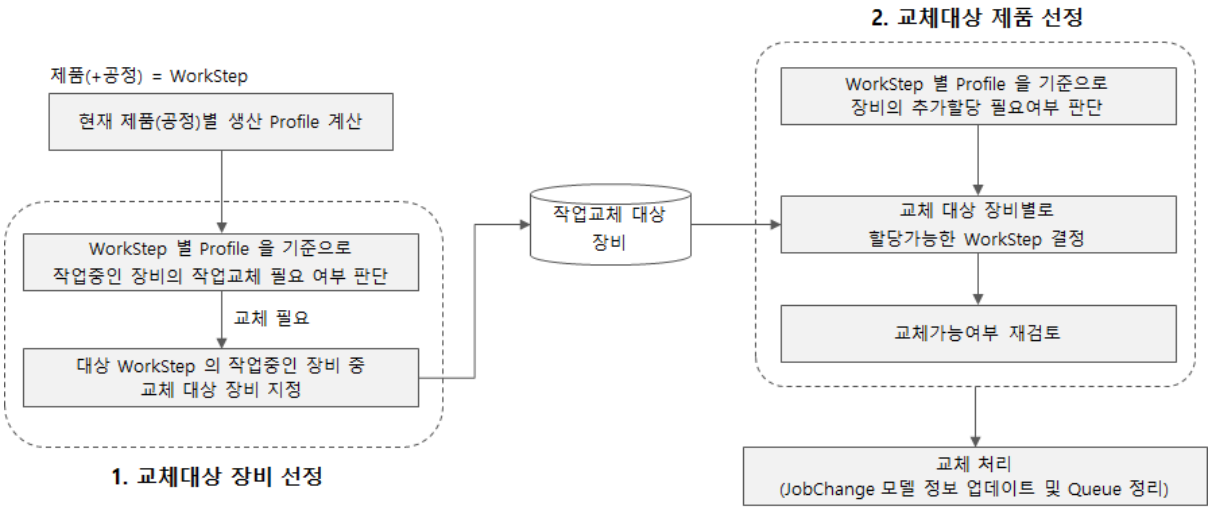
```
public void INITIALIZE_WORK_STEP_DEF(WorkStep wstep, ref bool handled)
{
    var agentInitControl = ServiceLocator.Resolve<JobChangeInit>();
    var loadables = agentInitControl.GetLoadableEqps(wstep);
    if (loadables != null)
    {
        loadables.ForEach(aeqp => wstep.AddLoadableEqp(aeqp));
    }
}
```

13. ***GetLoadableEqps*** : WorkStep 에 할당 가능한 설비의 목록을 설정합니다.
14. ***InitializeWorkEqp*** : WorkStep 에 LoadedEqp 를 추가할 때 WorkEqp 형식으로 저장이 되는데, 이 때 WorkEqp 의 초기화를 수행합니다.
15. ***IsRunningLot*** : WorkStep의 초기 할당 설비 설정할 때 사용자가 Run 재공 여부를 설정할 수 있습니다.
16. ***GetRunningWorkStep*** : IsRunningLot에서 판별된 Run 재공을 작업하 있는 WorkStep을 반환합니다.

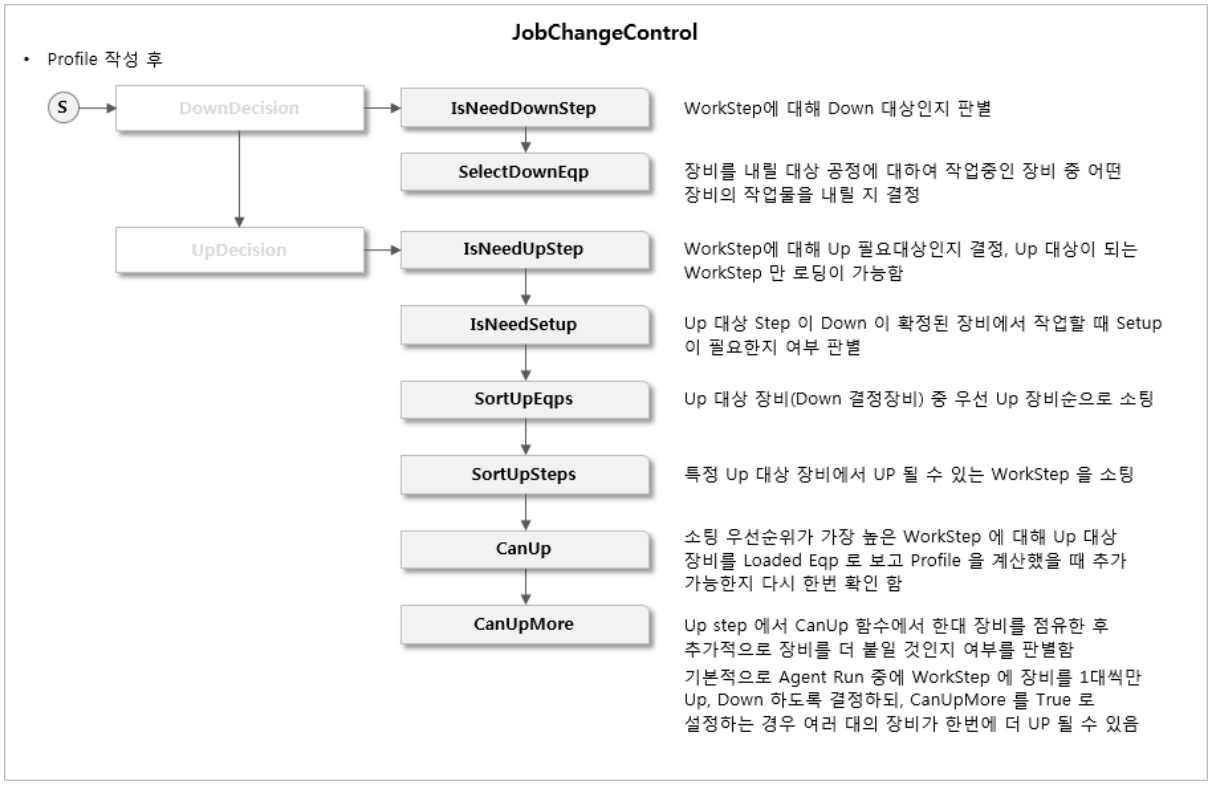
JobChangeControl 컨트롤

개요

JobChange 의사결정을 수행하고 Queue 에 대한 제어를 수행하는 JobChange Main Logic Component 입니다. WorkAgent 의 JobChange 의사 결정 방식 (AcgentType) 이 CHANGE 방식인 경우, 이 Component 가 실행됩니다. CHANGE 방식의 JobChange 의사 결정은 Down 대상 공정 (WorkStep) 을 우선 결정하고, Down 된 설비 또는 유휴 설비를 Up 대상 공정 (WorkStep) 에 할당하는 방식을 의미합니다. 이 Component 는 JobChange 의사 결정을 수행할 주기마다 호출됩니다.



컨트롤 설명



AgentControl Logic 은 크게 1) Down 대상 Step/장비 결정, 2) Up 대상 작업물 선택의 두 가지 Logic 을 수행하기 위한 함수들과 그 외 Agent 와 연계된 Simulation 객체를 제어하기 위한 추가적인 함수로 구성됩니다.

Down 대상 Step/장비 결정 : 계산된 WorkStep 별 Profile 를 입력으로 하여 기준에 따라 향후 장비를 내릴 WorkStep 을 결정하고 해당 WorkStep 을 로딩 중인 장비 중에서 작업교체 대상 장비를 선택합니다.

1. ***IsNeedDownStep*** : 대상 WorkStep 의 유입량 및 OutProfile 를 기준으로 대상 Step 의 작업중인 장비에 대한 작업교체가 필요한지 여부를 판별합니다. 이 때 WorkStep 에 설정된 Parameter 가 기본 Logic 에 사용되며, 필요 시 사용자 정의 Logic 으로 이를 선택할 수 있습니다.

   Default Logic Example

```
public bool IS_NEED_DOWN_STEP_DEF(WorkStep ws, ref bool handled, bool prevReturnValu
e)
{
   if (ws.LoadedEqpCount <= 1)
       return false;
   var now = ws.Agent.Now;
   var wlot = ws.FindOver(ws.DownInterval, ws.AllowedArrivalGap);
   if (wlot == null)
       return true;
   return wlot.InTime < now + ws.DownInterval;
}
```

2. ***SelectDownEqp*** : 선택된 Down 대상 WorkStep을 작업 중인 장비 중에 작업교체 대상 장비를 선택합니다.

   Default Logic Example

```
public IEnumerable<AoEquipment> SELECT_DOWN_EQP_DEF(WorkStep wstep, ref bool handle
d, IEnumerable<AoEquipment> prevReturnValue)
{
   var t = wstep.LoadedEqps.FirstOrDefault();
   return t != null ? new AoEquipment[] { t.Target } : null;
}
```

UP 대상 작업물 선택 : 유입량과 Out Profile 를 근거로 추가적으로 장비를 늘려야 하는 공정을 찾고, Down 대상 장비에서 작업교체 대상 공정을 선택합니다.

1. ***IsNeedUpStep*** : 대상 WorkStep 에 대해 유입량과 현재 작업중인 장비의 Out Profile 를 기준으로 추가적인 장비를 더 배정해야 하는지 여부를 판별합니다.

   Default Logic Example

```
public bool IS_NEED_UP_STEP_DEF(WorkStep ws, ref bool handled, bool prevReturnValue)
{
   if (ws.LoadableEqps.Count == 0)
       return false;
```

```
    if (!ws.UpableEqps().Any())
        return false;
    if (ws.LoadedEqpCount == 0 && ws.HasLoadableBeforeNextIteration())
        return true;
    var now = ws.Agent.Now;
    var wlot = ws.FindOver(ws.UpInterval, ws.AllowedArrivalGap);
    if (wlot == null)
        return false;
    return wlot.InTime >= now + ws.UpInterval;
}
```

2. ***IsNeedSetup*** : 작업 교체 대상 (Down) 장비에 대해 해당 장비에서 작업할 수 있는 UpStep 을 대상으로 작업교체가 필요한지 여부를 판별합니다.

3. ***SoutUpEqps*** : 작업 교체 대상 장비 중에 먼저 작업물을 선택할 장비 우선 순위에 따라 장비 (WorkEqp) 를 Sorting 합니다.

4. ***SortUpSteps*** : 특정 장비에 작업 교체 대상인 작업물 중 우선순위에 따라 작업물 (WorkStep) 을 Sorting 합니다. 우선순위가 높은 작업 (제품) 부터 장비에 먼저 로딩합니다.

5. ***CanUp*** : 대상 WorkStep 이 장비를 Up 시킬 수 있는지 여부를 판단합니다.

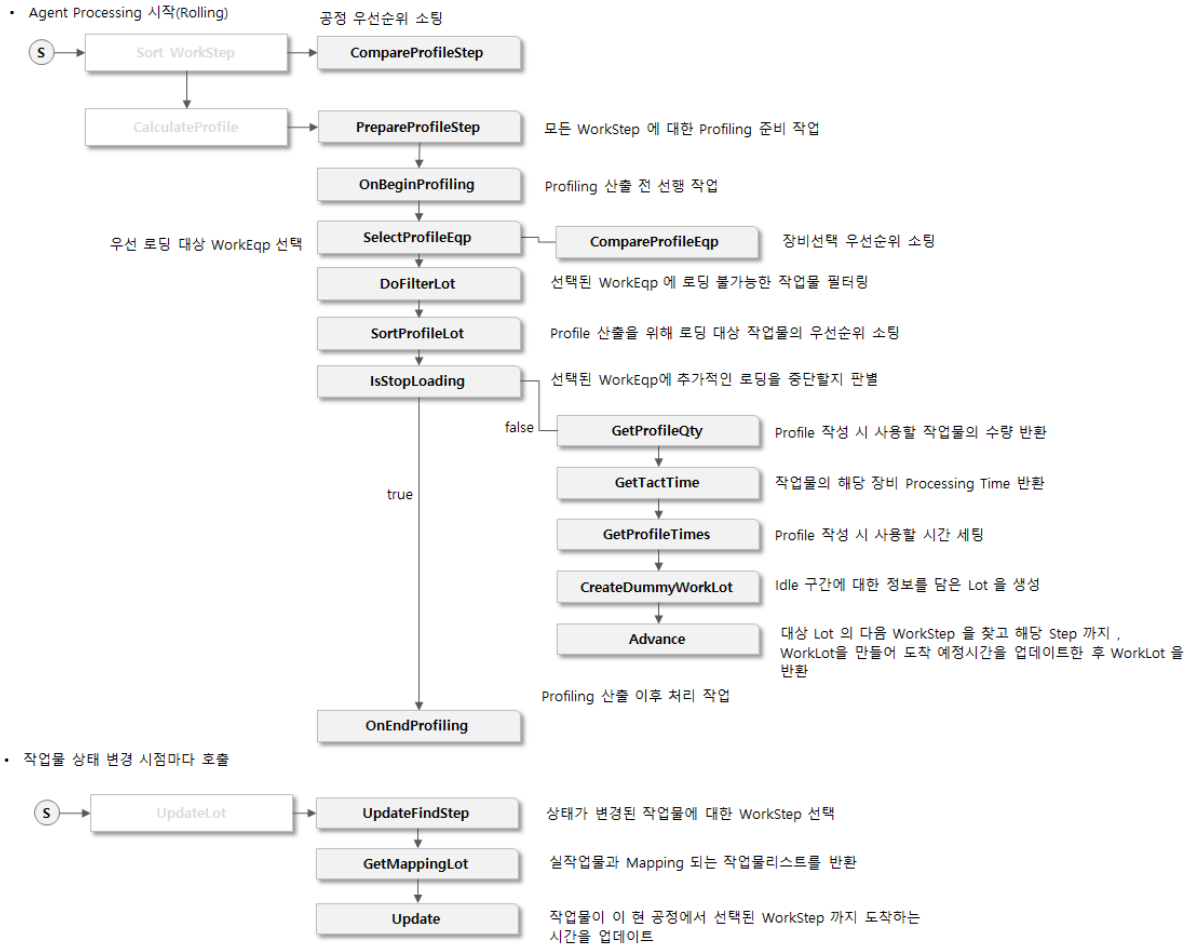
Default Logic Example

```
public bool CAN_UP_DEF(AoEquipment aeqp, WorkStep wstep)
{
    bool ret = false;
    try
    {
        wstep.AddLoadedEqp(aeqp);
        wstep.Group.CalculateProfile(wstep);
        var now = wstep.Agent.Now;
        var wlot = wstep.FindOver(wstep.NewUpInterval, wstep.AllowedArrivalGap);
        ret = wlot != null && wlot.InTime >= now + wstep.NewUpInterval;
    }
    finally
    {
        wstep.RemoveLoadedEqp(aeqp);
        if (!ret)
            wstep.Group.CalculateProfile(wstep);
    }
    return ret;
}
```

6. ***CanUpMore*** : Up 대상 Step 에 가용한 장비를 추가적으로 더 할당해야 하는지 여부를 판별 합니다.

JobProfileControl 컨트롤

JobProfileControl은 각 WorkStep 별 유입재공에 대한 장비 로딩을 통해 WorkStep 의 In/Out Profile 를 산출합니다.



1. ***CompareProfileStep*** : Profile 산출 전 동일한 WorkGroup 내의 WorkStep을 비교하여 정렬하기 위한 FEAction입니다. 본 FEAction은 WorkGroup.Ordered 속성이 true인 경우에만 호출됩니다.
2. ***PrepareProfileStep***: 모든 WorkStep에 대해 Profile 수행 전 선행 작업을 처리하기 위한 FEAction입니다. 현재 WorkStep에 작업중인 설비가 없는 경우 (`LoadedEqpCount` =0), `CompareProfileStep` FEAction 이후 `JobProfileControl` 내 어떤 FEAction도 호출되지 않습니다. 그래서 작업중인 설비가 없는 WorkStep에 대해 확인하거나 로그를 기록하는 로직을 `PrepareProfileStep` 에서 구현 할 수 있습니다.
3. ***OnBeginProfiling*** : Profile 산출 전 (Out Profile) 선행 처리를 위한 사용자 정의 로직을 구현하기 위한 FEAction입니다.

4. ***SelectProfileEqp*** : WorkStep 별로 유입재공을 처리할 대상 Loaded 장비를 찾습니다. 현재 WorkStep 을 작업중인 장비 중에 CompareEqp 에서 정의된 Sorting 순서로 장비를 찾아 반환하도록 Logic 이 구성됩니다.

   Default Logic Example

```
public WorkEqp SELECT_PROFILE_EQP_DEF(IEnumerable<WorkEqp> eqps, ref bool handled, W
orkEqp prevReturnValue)
{
   var jobChangeControl = ServiceLocator.Resolve<JobChangeControl>();
   WorkEqp min = null;
   foreach (var eqp in eqps)
   {
       if (eqp.IsDone())
           continue;
       if (min == null || jobChangeControl.CompareEqp(min, eqp) > 0)
           min = eqp;
   }
   return min;
}
```

5. ***CompareProfileEqp*** : 기본 Logic 은 가장 빨리 다음 작업을 로딩할 수 있는 장비 순으로 Sorting 입니다. 만일 사용자가 특정한 순서로 유입재공을 로딩하도록 조정하고자 하는 경우 해당 Sorting Logic 을 구현합니다. SelectProfileEqp 에 대해 직접적으로 재정의하는 경우 호출되지 않습니다.

   Default Logic Example

```
public int COMPARE_PROFILE_EQP_DEF(WorkEqp x, WorkEqp y, ref bool handled, int prevR
eturnValue)
{
   if (object.ReferenceEquals(x, y))
       return 0;
   int cmp = x.AvailableTime.CompareTo(y.AvailableTime);
   return cmp;
}
```

6. ***DoFilterLot*** : 실제 장비 (WorkEqp) 에 로딩해서 Out Profile 을 생성할 작업물을 선택합니다.

7. ***SortProfileLot*** : 실제 장비 (WorkEqp) 에 로딩해서 Out Profile 을 생성할 작업물을 정렬합니다.

   Default Logic Example

```
public void SORT_PROFILE_LOT_DEF(WorkStep wstep, List<WorkLot> list, ref bool handle
d)
{
```

```
    list.Sort(WorkLotComparer.Default);
}
```

8. ***IsStopLoading*** : 대상 Step 의 후속 Step 또한 Profile 계산 대상인 경우 false, 그렇지 않은 경우 true 를 반환하도록 Logic 을 개발합니다. 기본은 true 입니다.
9. ***GetProfileQty*** : ***IsStopLoading*** 의 반환 값이 false인 경우에 호출이 되며, Profile 작성을 위해 사용할 작업물의 수량을 반환하기 위한 FEAction입니다.
10. ***GetTactTime*** : ***IsStopLoading*** 의 반환 값이 false인 경우에 호출이 되며, Profile 작성을 위해 사용할 TactTime을 반환합니다.
11. ***GetProfileTimes***: Profile 작성에 필요한 시간 정보를 계산합니다. 시작 시간과 끝나는 시간이 동일한 유형은 Profile 작성에 반영되지 않습니다. 본 Action은 `WorkAgent.AddDummyProfile` 속성이 true로 설정된 경우에만 호출됩니다.
12. ***CreateDummyWorkLot***: 주어진 WorkLot의 프로파일 유형을 갖는 Dummy WorkLot을 생성합니다. 생성된 WorkLot의 InEqp, InTime, OutTime은 자동으로 설정됩니다. 이 Action은 `WorkAgent.AddDummyProfile` 속성이 true로 설정된 경우에만 호출됩니다.

    Default Logic Example

    ```
    public WorkLot CREATE_DUMMY_WORK_LOT_DEF(WorkStep wstep, WorkEqp weqp, WorkLot wlot,
    WorkLotType type, Time startTime, Time endTime, object stepKey, Step step, ref bool
     handled, WorkLot prevReturnValue)
    {
        return WorkLot.Create(null, startTime, stepKey, step, type);
    }
    ```

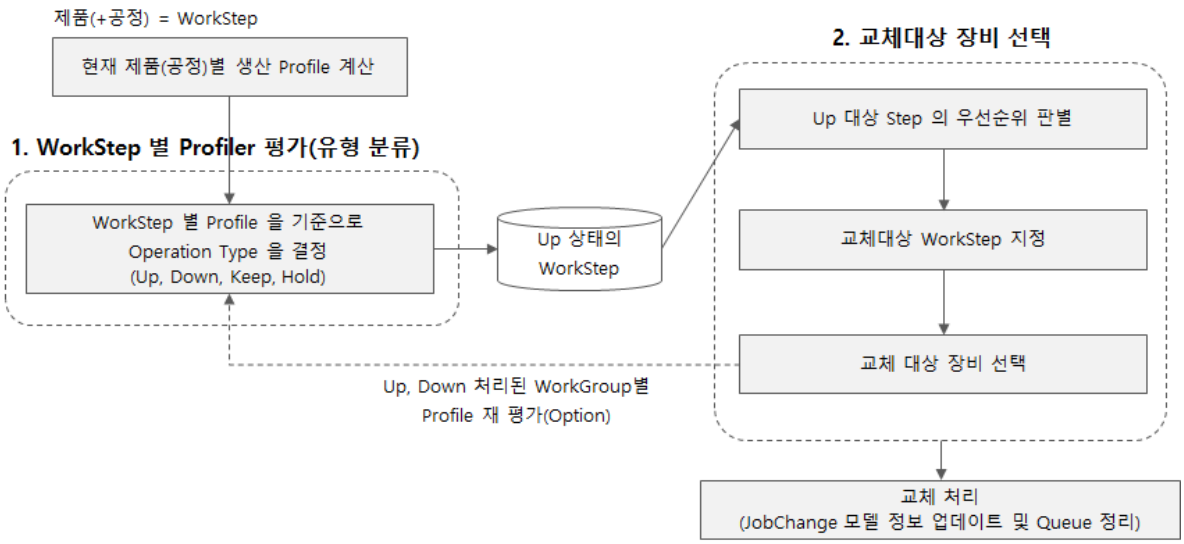
13. ***Advance*** : ***IsStopLoading*** 의 반환 값이 false인 경우에 호출이 되며, 현재 Step의 다음 Step 을 WorkLot 에 설정합니다.
14. ***OnEndProfiling*** : Profile 산출 후 (Out Profile) 후속 처리를 하기 위한 사용자 로직을 구현 할 수 있는 FEAction입니다.
15. ***UpdateFindStep*** : 상태가 변경된 작업물에 대한 WorkStep 을 선택합니다.
16. ***GetMappingLot*** : 실제 공정을 흐르는 작업물개체와 JobChangeAgent 에서 관리하는 작업물이 다른 경우 이를 Mapping 해주기 위한 FEAction. 예를 들어 실제 흐르는 작업물의 경우 WorkStep 에서 배치를 구성해서 흘러가더라도 JobChangeAgent 에서는 개별 Lot 으로 WorkLot 이 등록된 경우에 WorkLot 단위로 장비가 할당됨에 따라 실물 Batch Lot 이 로딩가능한 장비를 찾지 못하게 되거나, Batch 로 묶여 진행하는 Lot 들의 도착예정시간을 각 Lot 별로 업데이트하는 등의 작업을 하기 위해 필요합니다.
17. ***Update*** : JobChangeAgent 를 통해 JobChange 가 결정되는 재공, 즉 JobChange 대상 Step 의 유입재공으로 설정된 작업물은 상태 변경시점에 대상 Step 으로의 도착시간을

업데이트하여 대상 Step 의 In/Out Profile 을 좀 더 정확하게 계산할 수 있도록 합니다. 본 함수에서 대상 Step 의 유입재공에 대해 도착시간을 업데이트하여 반환합니다.

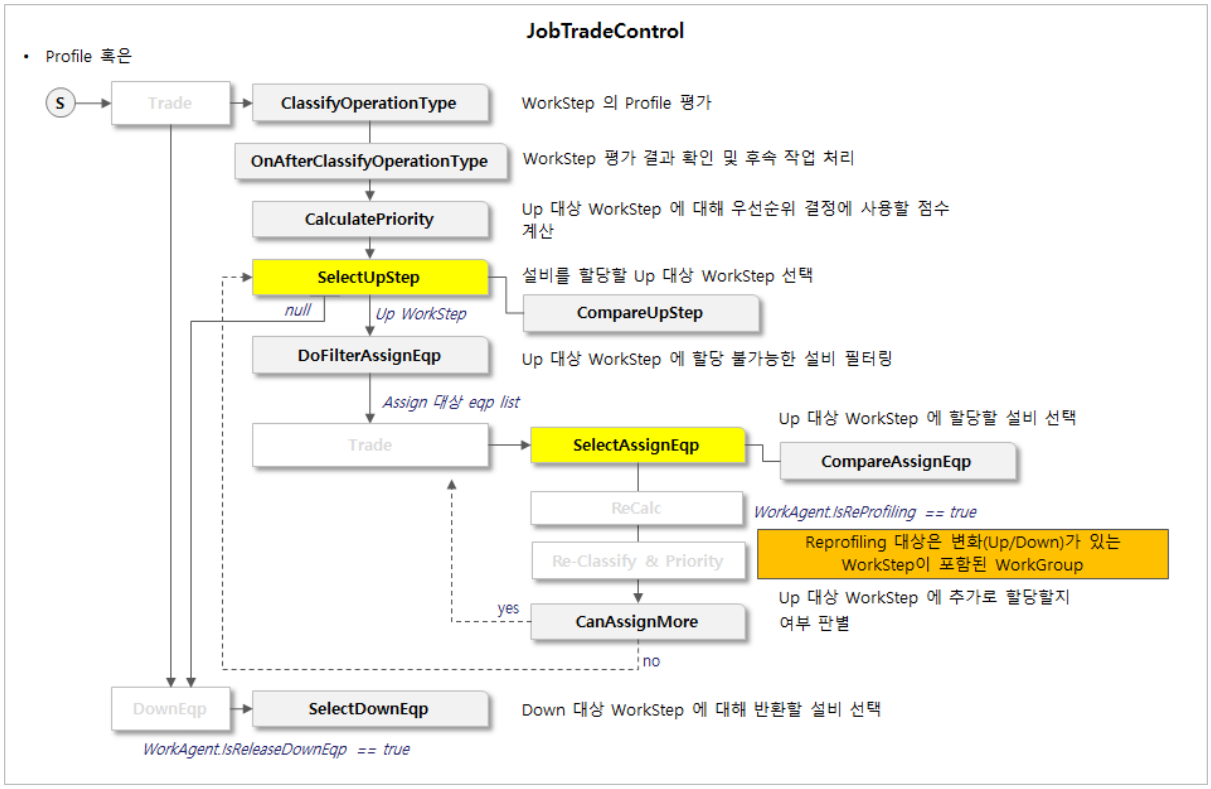
JobTradeControl 컨트롤

개요

JobChange 의사결정을 수행하고 Queue 에 대한 제어를 수행하는 JobChange Main Logic Component 입니다. WorkAgent 의 JobChange 의사 결정 방식 (AgentType) 이 TRADE 방식인 경우, 이 Component가 실행됩니다. TRADE 방식의 JobChange 의사 결정은 Up 대상 공정(WorkStep)을 우선 결정하고, 다른 공정(WorkStep)에 이미 할당된 설비나 유휴 설비를 Up 대상 공정 (WorkStep) 에 할당하는 방식을 의미합니다. 이 Component 는 JobChange 의사결정을 수행할 주기마다 호출됩니다.



컨트롤 설명



JobchangeTrade Logic 은 크게 1) Profile 평가 및 설비 재 할당, 2) 잔여 Down 대상 WorkStep 의 설비 반환 두 가지 로직을 처리하기 위한 함수들로 구성되어 있습니다.

1. ***ClassifyOperationType*** : WorkStep의 Profile을 평가합니다. 평가 결과는 Up/Down/Keep/Hold 로 구분됩니다. `ClassifyOperationType` 은 Profile 생성 후 한번 그리고 Up대상 WorkStep에 설비 할당 중 Reprofiling 하는 경우 (`WorkAgent.IsReprofiling=true`)일 때 `ReCalc` 과정에서 호출됩니다.
   - **Up** : 설비를 추가로 할당 받아야 합니다.
   - **Down** : 설비를 반환합니다. 설비를 한대 교체할 경우에도 Target을 달성할 수 있는 경우 Down 평가를 받습니다.
   - **Keep** : 현재는 Up 또는 Down 의 판단을 보류한 상태입니다.
   - **Hold** : 현재의 설비 상태를 유지하며, 현재 할당된 설비를 다른 WorkStep 에 빼앗기지 않습니다.
2. ***OnAfterClassifyOperationType***: WorkAgent의 실행 주기에 모든 WorkStep의 `OperationType`(Up, Down, Hold, Keep) 판단이 끝난 후 호출되는 FEAction으로 WorkStep 판단 결과를 조회하거나, WorkStep간 판단 결과를 비교하는 등의 로직을 구현할 수 있습니다. `ClassifyOperationType` 이후 호출되며, 초기 Profile에 대한 평가와 Up 대상 설비 할당 중 재평가가 필요한 경우에 (`WorkAgent.IsReporifling=true`) 호출됩니다. 이때 재평가 과정에서 OnAfterClassifyOperationType이 호출되면 `isReCalc` 파라미터 값은 `true` 입니다.

3. ***CalculatePriority*** : Up 대상 WorkStep 에 대해 우선순위를 결정합니다. `ClassifyOperationType` 함수에서 타입을 결정하기 위해 계산결과 값들을 `WorkStep.Data`에 저장한 후, 본 함수 호출 시 `WorkStep.Data`를 활용하여 Priority를 계산하는데 사용할 수 있습니다.

4. ***SelectUpStep*** : 우선적으로 설비를 할당할 Up 대상 WorkStep 을 선택합니다. 해당 FEAction을 별도로 구현하지 않는 경우 Default Definition에 의해 `CompareUpStep` 함수에서 우선순위가 가장 높은 Up 대상 WorkStep이 선택됩니다.

   Default Logic Example

```
public WorkStep SELECT_UP_STEP_DEF(List<WorkStep> upWorkSteps, JobChangeContext cont
ext, ref bool handled, WorkStep prevReturnValue)
{
   if (upWorkSteps.IsNullOrEmpty())
   return null;
   var jobChangeControl = ServiceLocator.Resolve<JobTradeControl>();
   upWorkSteps.QuickSort(jobChangeControl.CompareUpStep);
   return upWorkSteps.FirstOrDefault();
}
```

5. ***CompareUpStep*** : WorkStep 간 우선순위를 비교합니다. 기본 함수는 CalculatePriority FEAction에서 구현한 Priority를 사용하여 오룸차순 정렬합니다.

   Default Logic Example

```
public int COMPARE_UP_STEP_PRIORITY_DEF(WorkStep x, WorkStep y, ref bool handled, in
t prevReturnValue)
{
   return y.Priority.CompareTo(x.Priority);
}
```

6. ***DoFilterAssignEqp*** : 대상 WorkStep 의 로딩 가능 설비 중에서 현재 할당이 불가능한 설비를 필터링하고, 할당가능한 설비 리스트를 반환하는 함수입니다.

   - **assignEqps** = 현재 시점에 대상 WorkStep 의 로딩가능 장비 중 LoadedEqp 와 이번 Agent Run 중 Up 처리된 장비를 제외한 리스트를 반환합니다.

7. ***SelectAssignEqp*** : 대상 WorkStep 에 현재 할당 가능한 설비의 목록 중 할당할 설비를 하나 선택합니다.

   Default Logic Example

```
public List<AssignEqp> SELECT_ASSIGN_EQP_DEF(WorkStep upWorkStep, List<AssignEqp> as
signEqps, JobChangeContext context, ref bool handled, List<AssignEqp> prevReturnValu
```

```
e)
{
   if(assignEqps.IsNullOrEmpty())
   return null;
   var jobChangeControl = ServiceLocator.Resolve<JobTradeControl>();
   assignEqps.QuickSort(jobChangeControl.CompareAssignEqp);
   return new List<AssignEqp>() { assignEqps[0] };
}
```

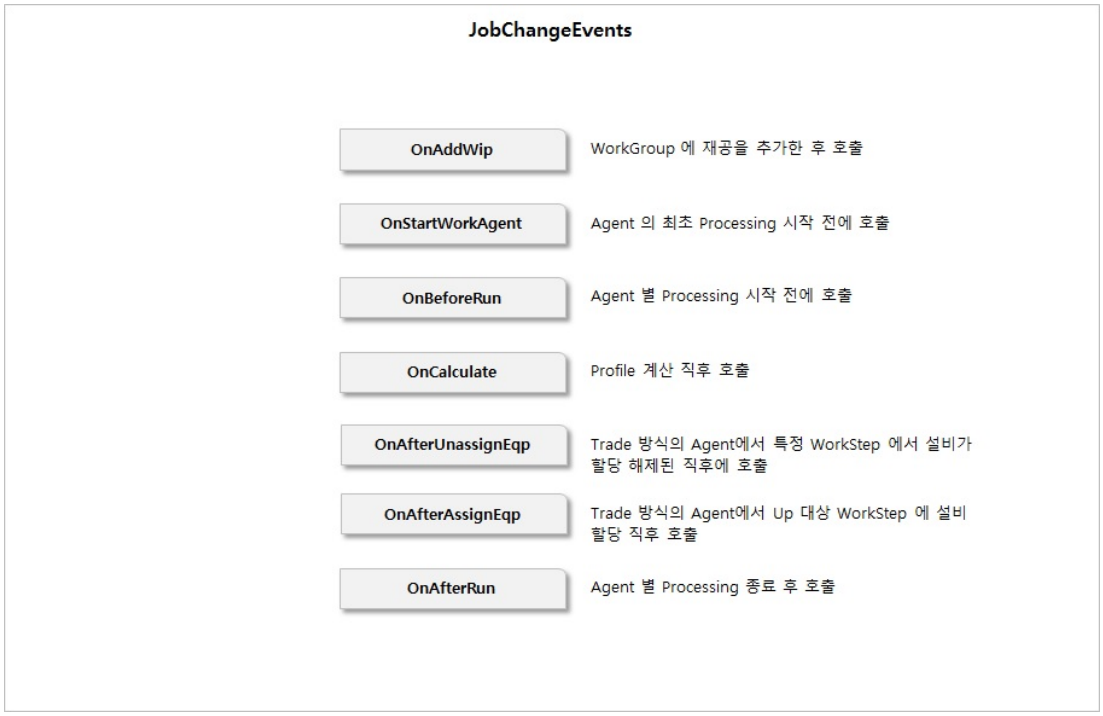
8. ***CompareAssignEqp*** :할당가능한 설비의 우선순위를 비교합니다.
9. ***CanAssignMore*** : 대상 Workstep 에 설비를 할당한 뒤에 추가로 설비를 할당할지 여부를 판별합니다.
10. ***SelectDownEqp*** : Down 대상 WorkStep 에 대해 반환할 설비의 목록을 선택합니다. 선택된 설비는 유휴 설비가 됩니다.

    Default Logic Example

```
public IEnumerable<AoEquipment> SELECT_DOWN_EQP_T_DEF(WorkStep wstep, ref bool handl
ed, IEnumerable<AoEquipment> prevReturnValue)
{
   var t = wstep.LoadedEqps.FirstOrDefault();
   return t != null ? new AoEquipment[] { t.Target } : null;
}
```

JobChangeEvents 컨트롤

JobChangeAgent Model 의 실행시점에 발생하는 Event 처리 함수 집합입니다. 사용자 정의 Logic 을 해당하는 시점에 추가할 수 있습니다.



1. ***OnAddWip*** : WorkStep 에 유입대상 재공을 추가한 후 호출되는 Event 입니다.
2. ***OnStartWorkAgent*** : Agent 최초 Logic 처리 시작 전에 호출되는 Event 입니다. 부족한 초기화 Logic 을 추가할 수 있습니다.
3. ***OnBeforeRun*** : AgentControl Logic 수행 전에 호출되는 Event 입니다.
4. ***OnCalculate*** : AgentControl 의 Profile 계산 직후 호출되는 Event 입니다.
5. ***OnAfterUnAssignEqp***: TRADE 방식의 Agent에서 사용됩니다. Job Change 대상 공정의 작업 가능한 설비에서 제외될 때 호출되는 Event 입니다.
6. ***OnAfterAssignEqp*** : TRADE 방식의 Agent 에서 Up 대상 WorkStep 에 설비를 할당한 직후에 호출되는 Event 입니다.
7. ***OnAfterRun*** : AgentControl Logic 수행이 완료된 직후에 호출되는 Event 입니다.

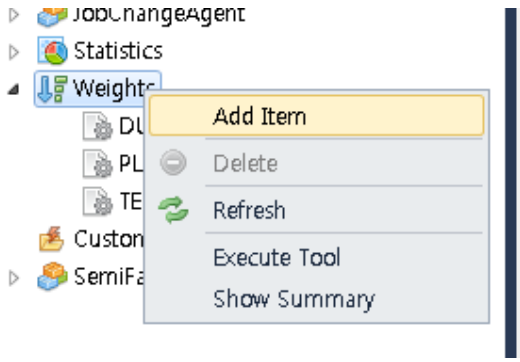
Weight Factor Method 개발

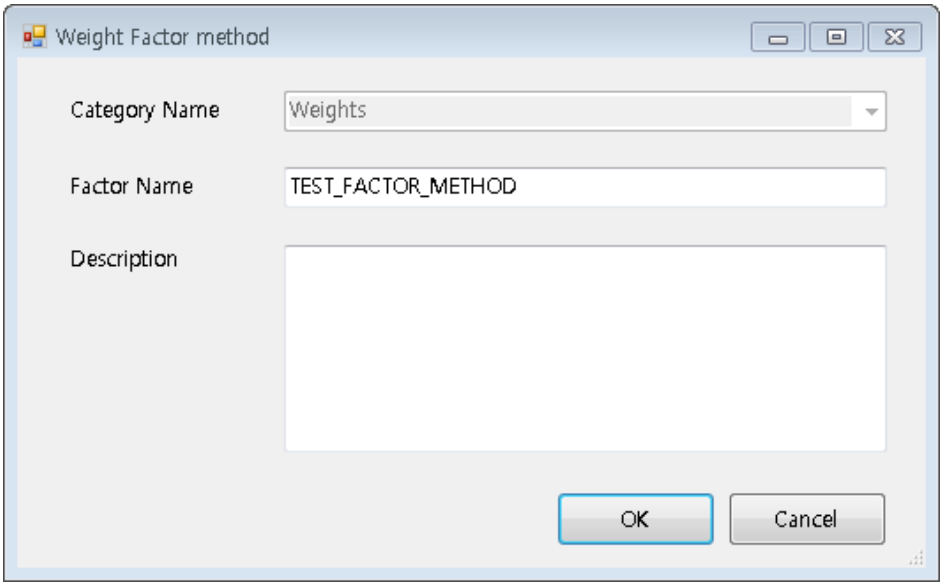
Simulation 라이브러리에서 제공하는 Weight Factor 기반의 Dispatcher를 사용할 경우 대상 구축시스템의 Dispatching 방식을 반영하기 위해 Weight Factor의 평가를 위한 Method를 생

성할 수 있습니다. Weight Factor Method는 Weigt Factor 에 대한 점수산출 로직을 구현한 함수입니다. Simulation 모델 구현시 해당 모델에서 사용할 Dispatching 방식을 결정하고, 장비별로 사용할 Weight Preset 을 설정하면 장비 Dispatching 시점에 설정된 Preset 을 확인하고, Preset을 구성하는 Weight Factor 정보와 Factor Method 를 사용하여 재공의 우선순위를 평가합니다. 보다 상세한 내용은 **SeePlan Dispatcher Concept** , **Weight Preset 기반 Dispatcher 구축방법**을 참조하시기 바랍니다.

Weight Factor Method를 추가하기 위해서는 다음의 절차를 수행합니다.

1. 모델 탐색기에서 Simulation/Weights 노드를 선택하고 ***Add Item*** 버튼을 누릅니다.





2. 새로 작성할 Weight Factor Method 명을 입력합니다. Weigh Factor 명은 Input 으로 정의할 Preset 의 factor 명을 동일하게 설정합니다.
3. Method는 Weights Category로 자동 등록됩니다.

4. Method를 정보 입력이 완료되면 솔루션 탐색기의 Logic 메뉴 하단에 Weights 클래스를 생성합니다. 이미 존재하는 경우 Method만 등록하며, 기존에 Weights 클래스가 없는 경우에만 생성합니다.
5. Method 원형을 자동이 자동 생성되면 Method를 편집할 수 있도록 해당 Method가 포함된 .cs 파일을 소스편집기에 열리고, 커서가 해당 함수 위치합니다. 함수의 Return value 는 WeightValue type 으로 실제 계산 결과와 Value 에 대한 Description 으로 구성됩니다. Value 는 실제 라이브러리상에서 데이터 소팅시 사용되며, Description 의 경우 향후 Dispatching log 분석 시 사용될 수 있습니다. Dispatching log 기록 할때 값만으로 의미가 명확하지 않은 경우 **Dispatching>DispatcherControl>AddDispatchWipLog** 함수 WeightValue 의 Description을 Value 대신 기록하여 사용할 수 있습니다.

   Weight method code 작성

   ```
   public WeightValue HOT_PRIORITY(ISimEntity entity, DateTime now, AoEquipment eqp, We
   ightFactor factor, IDispatchContext ctx)
   {
      var lot = CommonHelper.GetLot(hb) ;
      float value = DisptchHelper.GetHotValue(lot.HotType) ;
      value = value * factor.Factor;
      return new WeightValue(value, lot.HotType.ToString() );
   }
   ```

6. 함수 수정이 완료되면 Weight Factor의 사용준비가 완료됩니다. 이후에는 해당 Factor 를 Preset 정보에 추가하여 실 테스트를 수행할 수 있습니다.

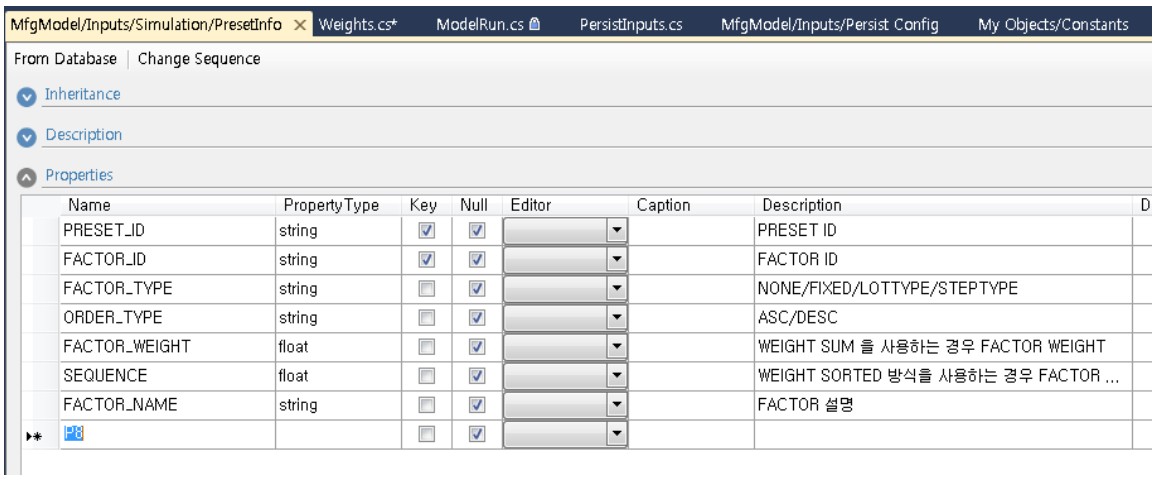
Weight Preset 기반 Dispatcher 구현방안

Weight Preset 기반의 Dispatcher 를 정의하기 위해서는 아래의 절차를 따릅니다.

1. WeightPreset 정보를 Input Data 로 만듭니다. Input Data의 형식은 다음과 같습니다.
   - PRESET_ID : PRESET ID 입니다. 장비의 PRESET 정보 MAPPING 시에도 사용되는 PRESET Identifier 입니다.
   - FACTOR_ID : Weight Factor 에 대한 ID 입니다. Simulation 기능 중 Weights 를 생성할 때 사용하는 Factor Method 명칭과 동일해야 합니다. (**Weight Factor Method 추가 방법**)
   - FACTOR_TYPE : FACTOR VALUE 계산시 중복적인 계산을 방지하기 위해 FACTOR VALUE 의 계산 단위를 설정합니다.

    - FIXED = LOT, STEP 과 상관없이 DISPATCHING 시점에 모든 LOT 에 동일한 값으로 계산되는 값인 경우
    - LOTTYPE = LOT 별로 점수가 계산되어야 하는 FACTOR (예: DUE_DATE, ..)
    - STEPTYPE = STEP별로 점수가 계산되어야 하는 FACTOR (예: STEP TARGET 달성율, ..)
- ORDER_TYPE : WeightSorted Dispatcher 에서 사용되는 속성, Factor 소팅시 소팅 방식
    - ASC = 오름차순
    - DESC = 내림차순
- SEQUENCE : WeightSorted Dispatcher 에서 사용되는 속성, Factor 소팅 우선순위 입니다. (Factor 순서에 따라 Factor Value 로 소팅을 실행하며, 동점인 Lot 이 있는 경우에는 다음 순위의 Factor Value 로 순서대로 소팅을 합니다.
- FACTOR_WEIGHT : WeightSum Dispatcher 에서 사용되는 속성, Factor Value 의 가중합을 계산할때 사용되는 Factor 가중치입니다.
- FACTOR_NAME : FACTOR 설명, 명칭입니다.

[구현 예]



Name	PropertyType	Key	Null	Editor	Caption	Description
PRESET_ID	string	☑	☑			PRESET ID
FACTOR_ID	string	☑	☑			FACTOR ID
FACTOR_TYPE	string	☐	☑			NONE/FIXED/LOTTYPE/STEPTYPE
ORDER_TYPE	string	☐	☑			ASC/DESC
FACTOR_WEIGHT	float	☐	☑			WEIGHT SUM 을 사용하는 경우 FACTOR WEIGHT
SEQUENCE	float	☐	☑			WEIGHT SORTED 방식을 사용하는 경우 FACTOR ...
FACTOR_NAME	string	☐	☑			FACTOR 설명
		☐	☑			

[데이터 예]

In/Simulation/PresetInfo *

Open Schema | Open Action | Test Values | Query & Save to file | Fix Data

Description

Drag a column header here to group by that column

PRESET_ID	FACTOR_ID	FACTOR_TYPE	ORDER_TYPE	FACTOR_WEIGHT	SEQUENCE	FACTOR_NAME
P1	DUE_DATE	LOTTYPE	ASC	0.5	1	
P1	PLAN_RATE	STEPTYPE	ASC	0.5	2	
P2	PLAN_RATE	STEPTYPE		0.8		
P2	DUE_DATE	LOTTYPE		0.2		

2. 장비정보에 장비가 사용할 Preset 과 Dispatching 방식을 설정합니다. 아래는 설정 예입니다.

Experiment 1 * | In/Simulation/Equipment

Open Schema | Open Action | Test Values | Query & Save to file | Fix Data

Description

Drag a column header here to group by that column

LINE_ID	EQP_ID	PROCESS_GROUP	SIM_TYPE	PRESET	DISPATCHER_TYPE	MIN_BATCH	MAX_BATCH	PROCESS_CAPA
LINE01	M01	MILLING	Table	P1	WeightSum			
LINE01	M02	MILLING	Table	P1	WeightSum			
LINE01	C01	CLEAN	Inline	P1	WeightSum			
LINE01	E01	EDM	LotBatch	P1	WeightSum	1	2	

3. 장비정보 로딩시 상기정보를 설정합니다.

장비 정보 로딩구현 예시

```
// 1. 장비정보 로딩
public bool OnAfterLoad_Equipment(Equipment entity)
{
   SimpleMfgSemiconEqp eqp = entity.ToSimpleMfgSemiconEqp();

   LocationInfo loc = new LocationInfo();
   loc.LineID = entity.LINE_ID;

   eqp.Init(entity.EQP_ID,
   DateTime.MinValue,
   DateTime.MaxValue,
   entity.SIM_TYPE,
   loc
   );

   eqp.DispatchingRule = entity.DISPATCHER_TYPE;
   eqp.Untilization = 1.0f;
```

```
   WeightPreset preset ;
   if (InputMart.Instance.PresetList.TryGetValue(entity.PRESET, out preset))
   {
       eqp.Preset = preset;
   }

   eqp.EqpDispatchInfo = new EqpDispatchLogInfo(eqp);
   InputMart.Instance.SimpleMfgSemiconEqp.Add(eqp.EqpID, eqp);

   return true;
}
// 2. Preset 정보 로딩
public void OnAction_Preset(IPersistContext context)
{
   InputMart.Instance.PresetInfo.DefaultView.Sort = "PRESET_ID ASC, FACTOR_ID ASC ";

   WeightPreset preset;
   foreach (PresetInfo wp in InputMart.Instance.PresetInfo.DefaultView)
   {
       if (InputMart.Instance.PresetList.TryGetValue(wp.PRESET_ID, out preset) == fa
lse)
           InputMart.Instance.PresetList.Add(wp.PRESET_ID, preset = new WeightPreset
(wp.PRESET_ID));

       WeightFactor factor = new WeightFactor(wp.FACTOR_ID, wp.FACTOR_WEIGHT,
                   wp.SEQUENCE, (FactorType)Enum.Parse(typeof(FactorType),
                   wp.FACTOR_TYPE),(OrderType)Enum.Parse(typeof(OrderType),
                   wp.ORDER_TYPE));

       preset.FactorList.Add(factor);
   }
}
```

4. 장비 초기화 및 Preset 에 대한 초기화 코드를 작성합니다. GetDispatcherType FEAction 의 구현시에 결과 Key 의 종류는 "Fifo", "WeightSum", "WeightSorted" 의 세 가지이며, FifoDispatcher 설정시에는 Preset 에 대한 설정은 필요없습니다. Queue 에 들어가는 순서대로 작업을 진행합니다.

   초기화 코드 작성 예시

```
// 1. 장비초기화 (Factory>EqpInit>GetDispatcherType 의 구현)
// Dispatcher Type Key = {"Fifo", "WeightSum", "WeightSorted"}
public string GETDISPATCHERTYPE_0(Resource eqp, ref bool handled, string prevReturnV
alue)
{
    SimpleMfgSemiconEqp mEqp = eqp as SimpleMfgSemiconEqp;
    return mEqp.DispatchingRule;
}
// 2. Preset 초기화 (Factory>FactoryInit>GetWeightPreset 의 구현)
public IEnumerable<Mozart.SeePlan.DataModel.WeightPreset> GETWEIGHTPRESETS_0(ref boo
l handled, IEnumerable<Mozart.SeePlan.DataModel.WeightPreset> prevReturnValue)
{
    return InputMart.Instance.PresetList.Values.ToList();
}
```

5. 사용된 Factor Value 를 계산하는 Weights 를 추가합니다. (**Weight factor method 개발 참조**)

Preset 기반 Dispatcher Concept

Mozart SeePlan 라이브러리는 Weight Factor 와 Factor 의 Preset 을 사용한 두가지 Dispatcher 를 제공합니다. Dispatching 시 사용되는 용어의 의미는 다음과 같습니다.

Weight Factor

작업물의 우선순위를 평가하기 위한 평가요소를 의미합니다. 작업물을 선입선출 방식으로 Dispatching 하는 경우 이를 작업물 단위로 공정에 도착한 시간으로 비교할 수 있도록 Weight Factor를 정의할 수 있습니다. 통상 하이테크 산업의 제조현장에서 Dispatching 시 작업우선순위를 평가하는 평가요소는 적게는 3~4개 많게는 10~20여 가지로 구성됩니다.

Weight Preset

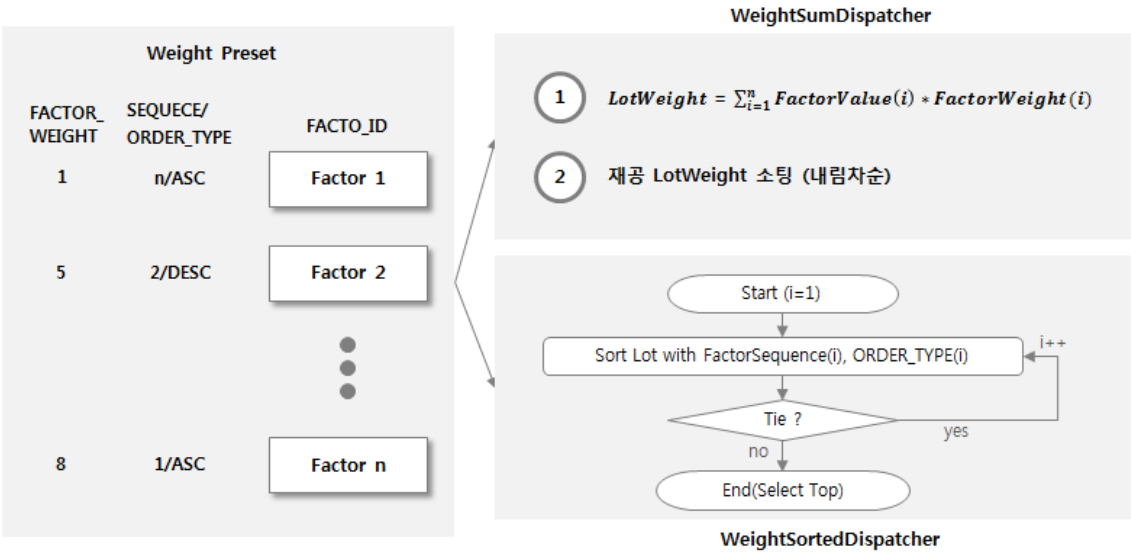
Weight Preset 은 Dispatching 시 사용되는 Weight Factor 의 집합이며, Mozart 에서는 Factor 간 우선순위 혹은 Dispatcher Type 에 따른 Factor 속성정보를 포함하는 데이터 Set 를 의미합니다. Dispatching 방식은 장비에 따라 다른 방식으로 진행 할 수 있음으로 실제 사이트에서 사용하는 Preset 은 1개에서부터 장비타입별로 다양하게 구성될 수 있습니다. 또한 장비의 기준정보로 장비가 사용할 Preset 과 Dispatcher 에 대한 설정이 필요합니다.

Factor Method

Weight Factor 에 대한 평가 결과 값을 산출하기 위한 함수를 의미합니다. Factor 의 값을 산출하기 위해서는 작업물정보, 장비정보, Preset 에 포함된 Factor 의 속성정보가 필요합니다. Dispatcher 의 적용시 Preset 을 구성하는 Weight Factor 에 대한 Factor Method가 반드시 구현되어 있어야 하며 Mozart 라이브러리의 Simulation Module 에서는 **Weights 기능**을 통해 Factor Method를 정의할 수 있습니다.

Dispatcher

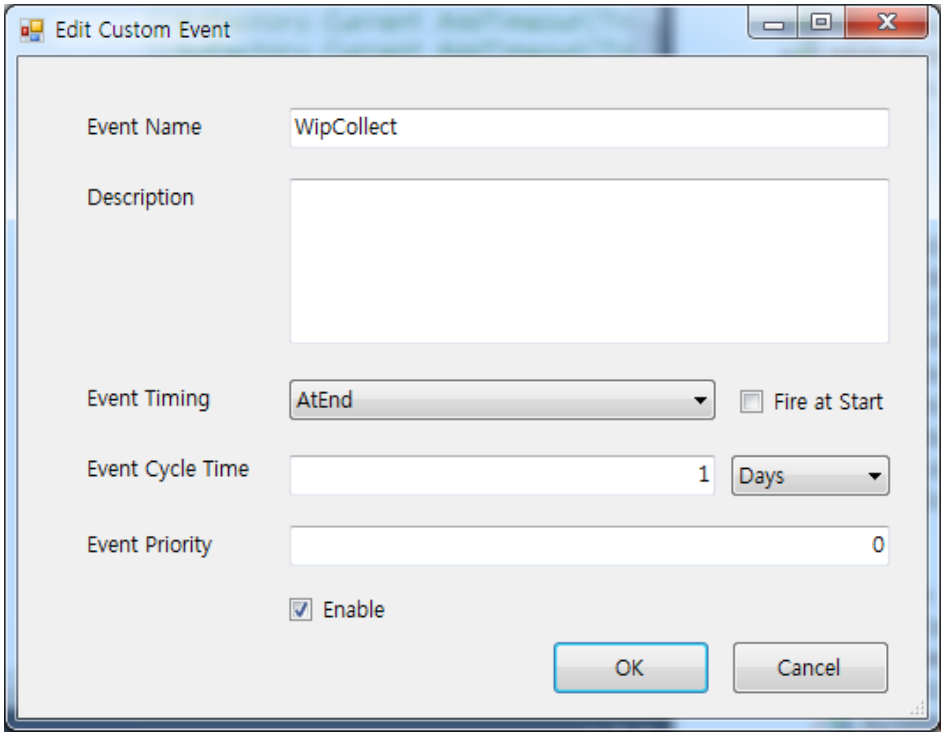
Dispatcher 는 장비에서 작업물을 선택할 때 대기중인 작업물을 장비의 Weight Preset 에 포함된 Factor 별로 평가점수를 산출한 후 Dispatching Type에 따라 작업물의 우선순위를 평가하여 작업대상물을 선택하는 역할을 수행합니다. Mozart 에서는 Preset 을 구성하는 Factor 별 점수의 가중합 방식(WeightedSumDispatcher)과 Factor별 점수의 순차정렬방식(WeightSortedDispatcher) Dispatcher 두가지를 지원합니다. 아래 그림은 두가지 방식의 개념을 설명합니다.



Custom Event 개발

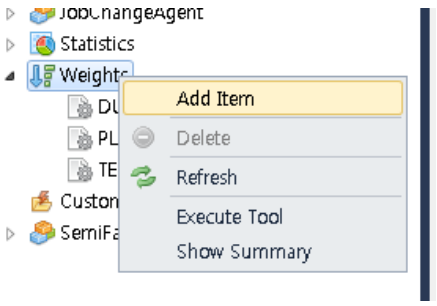
Simulation Core Model 의 Event 외에 사용자가 특정 시점, 특정 주기로 Event 를 발생시켜 사용자 정의 Logic 을 실행시켜야 하는 경우 Custom Event 를 정의할 수 있습니다. Custom Event 를 정의하기 위해서는 다음의 절차를 따릅니다.

1. **[Simulation > CustomEvents]** 노드에서 팝업메뉴로 **[Add]** 버튼을 클릭합니다.
2. 'Add Custom Event' 다이얼로그를 통해 Event 속성을 입력합니다

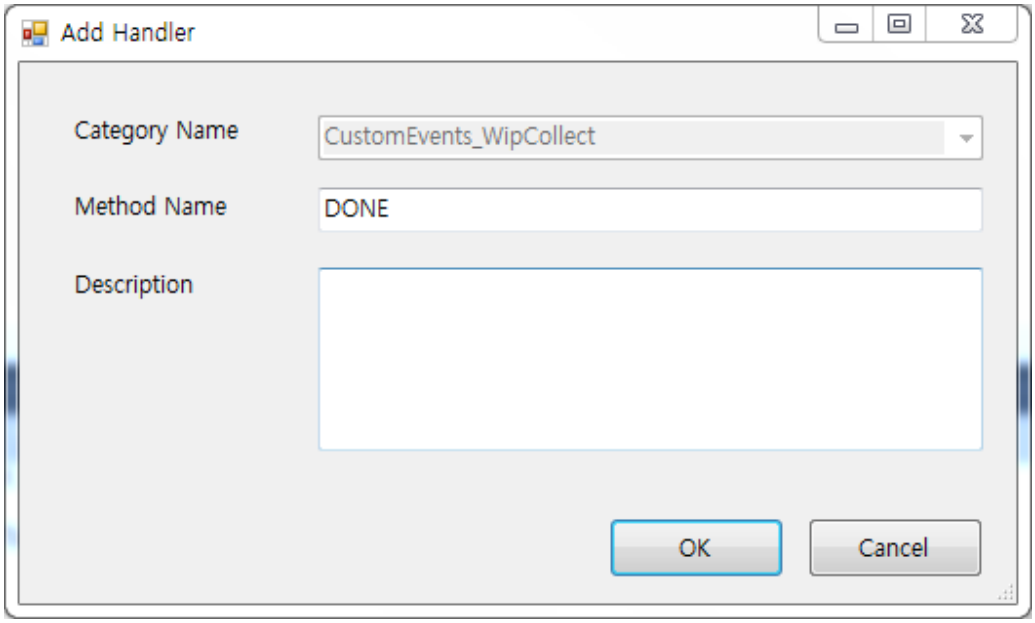


- **Event Name** : Event 명을 입력합니다.
- **Description** : Event 에 대한 설명을 입력합니다.
- **Event Type** : 동일 시간 발생 Event 와의 선후 관계를 설정 할 수 있습니다.
  - **AtBegin** : 동일 시간 발생 Event 들을 처리하기 전에 가장 먼저 처리하는 옵션
  - **AtEnd** : 동일 시간 발생 Event 들을 모두 처리한 후 처리하는 옵션
- **Fire at Start** : Simulation 시작 시점에 첫 Event 를 처리할 것인지 여부입니다. 옵션이 설정되지 않으면 Simulation 시작부터 Event Cycle Time 이후에 첫 Event 가 발생합니다.
- **Event Cycle Time** : Event 수행 주기를 입력합니다. 이 때 주기는 일, 시간, 분 단위로 선택하여 입력할 수 있습니다.
- **Event Priority** : Event Type 이 동일한 Custom Event 가 다수 존재할 때 실행 우선순위입니다. 순서는 오름차순입니다. 즉 낮은 수가 우선이며 음수도 설정 가능합니다.
- **Enable** : Event 활성화 여부입니다. Enable 체크 해제된 경우 Event 가 실행되지 않습니다.

3. Event 속성을 입력하고 **[OK]** 버튼을 클릭하면 MOZART Explorer 의 CustomEvents 노드 밑에 입력한 Event 명으로 신규 Event 가 생성됩니다.
4. Custom Event 에 대해 초기화 및 Event 처리 함수를 정의할 수 있습니다.



5. 각 함수의 구현을 위해 함수를 더블 클릭하거나 팝업메뉴의 [Edit] 버튼을 클릭하면 해당 함수를 정의할 수 있는 'Add Handler' 다이얼로그가 로드되고 설명을 입력할 수 있습니다.



- **Category Name** : Event 에 대한 자동 생성 클래스 명으로 자동 설정되며 변경이 불가합니다.
- **Method Name** : 기본적으로 Method 명으로 생성되며 변경이 가능합니다.
- **Description** : Method 설명을 입력합니다.

6. 처리함수의 속성을 입력하고 [OK] 버튼을 클릭하면 Solution Explorer 의 [Logic > Simulation] 폴더 하위에 해당 Event 에 대한 클래스파일이 생성됩니다. 파일명은 소스 편집기에 해당 Method 를 편집할 수 있도록 커서가 위치합니다.

```
public bool RUN(Mozart.SePlan.Simulation.ICalendarEvent evt, ICalendarEventManager
 cm)
{
  // Event 처리 Logic
  // ....

  // 반복적인 Event 를 생성하기 위해 true 반환
```

```
  return true;
}
```

7. 함수의 편집을 수행하면 Custom Event 개발이 완료됩니다.

Filter Method 개발

장비/작업물 관점에 따라 Dispatching 제약에 의해 작업물 Filtering이 필요로 하는 경우 Filtering 조건을 정의 할 수 있는 Method를 생성 할 수 있습니다. 제약 조건들이 다수가 있을 경우 Filter에 대한 Key 값을 설정하고 Key에 해당하는 Filter들을 등록하여 FilterSet을 생성 할 수 있습니다. 본 장에서는 Filter Key, Filter Set 및 Filter Method 구현 방식을 설명합니다.

Filter Key 및 Filter Set 생성하기

Dispatching 제약을 위한 Filter Key 및 Filter Set을 Simulation 중 Factory 초기화 시점에 사용자가 로직을 구현하여 FilterManager에 등록을 합니다..
Simulation/Factory/FactoryInit/InitializeFilterManager를 통해 로직을 구현합니다. 본 예제에서는 Input으로 Filter 정보를 가지고 있고 장비 Input 정보에는 Preset과 같이 FACTOR_ID를 설정한 Scenario의 예제입니다.

`<<Filter Input>>`

FILTER_ID	FILTER_FUNC
FILTER01	IS_NOT_TARGET_HOLD
FILTER01	CHECK_WEIGHT_FILTER
FILTER02	IS_LOADABLE_CHAMBER

`<<Equipment Input>>`

LINE_ID	EQP_ID	EQP_GROUP	SIM_TYPE	DISPATCHER_TYPE	PRESET_ID	STATUS	STATUS_CHANGE_TIME	FILTER_ID
LINE01	EQP01	EG01	TABLE	FIFO	PRESET01	UP	2016-10-19 05:00:00	FILTER01
LINE01	EQP04	EG03	TABLE	FIFO	PRESET01	UP	2016-10-19 05:00:00	FILTER01
LINE01	EQP05	EG03	TABLE	FIFO	PRESET01	UP	2016-10-19 05:00:00	FILTER01
LINE01	EQP06	EG03	TABLE	FIFO	PRESET01	UP	2016-10-19 05:00:00	FILTER02

Sample Code

```
public void INITIALIZE_FILTER_MANAGER0(IFilterManager filterManager, ref bool ha
ndled)
{
    string Key = null;
    List<string> value = new List<string>();

    var filterType = InputMart.Instance.Filters.Rows.GroupBy(x => x.FILTER_ID).S
elect(group => new { Key = group.Key, value = group.ToList() });
    foreach (var item in filterType)
    {
        // FilterManager에 Key와 Filter Factor를 등록합니다.
        filterManager.CreateMethods(item.Key, item.value.Select(x => x.FILTER_FU
NC));
    }
}
```

FilterControl에서 처리

Simulation/Factory/FactoryInit/InitializeFilterManager를 통해 Filter Key와 Filter들을 FilterManager에 등록을 하였으면, Simulation/FilterControl/GetFilterSetKey에 Filtering에 사용할 Filtering Set의 Key를 반환합니다. Filter Set의 각 Filter들은 Do Filter 시점에 호출이 됩니다. 본 예제는 장비 정보에 설정된 Filter ID를 반환합니다.

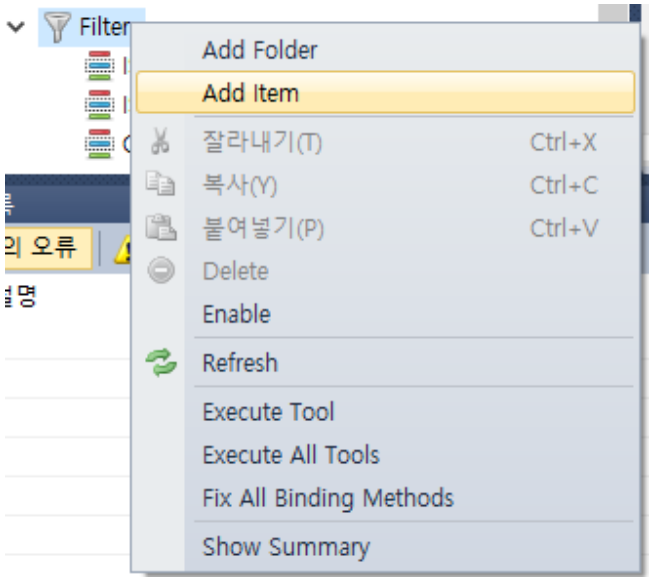
Sample Code

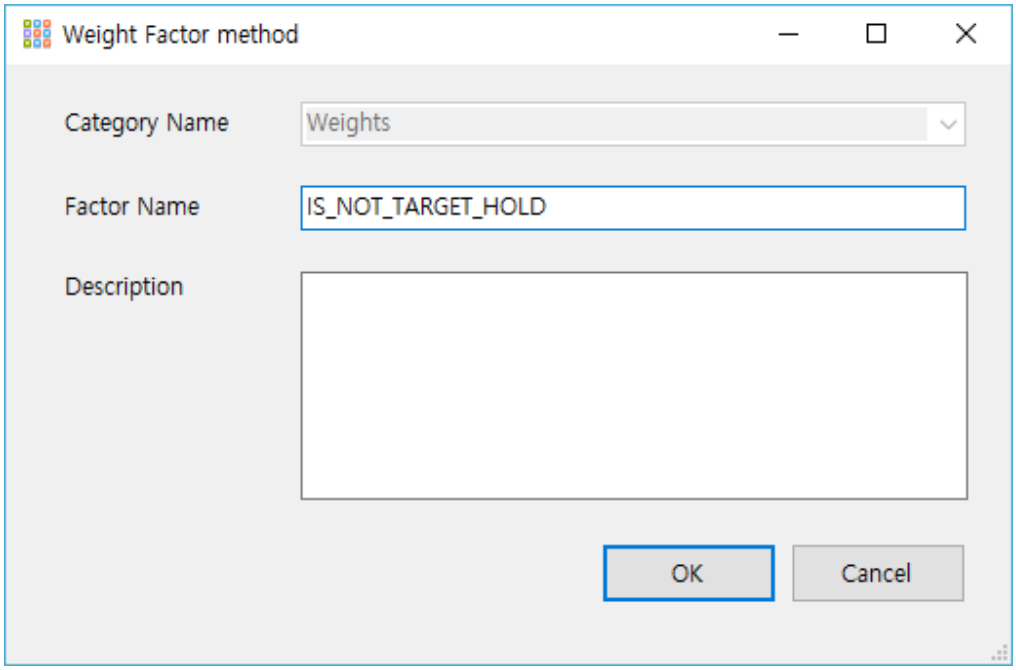
```
/// <summary>
///
/// <param name="eqp"/>
/// <param name="hb"/>
/// <param name="ctx"/>
/// <param name="handled"/>
/// <param name="prevReturnValue"/>
/// <returns/>
public string GET_FILTER_SET_KEY0(Mozart.SeePlan.Simulation.AoEquipment eqp, IHa
ndlingBatch hb, IDispatchContext ctx, ref bool handled, string prevReturnValue)
{
    VmsEqp resource = eqp.Target as VmsEqp;
    return resource.FilterID;
}
```

Filter Method 구현

Simulation/FilterControl/DoFilter에서 Filter Set Key에 해당하는 Filter를 호출 시 해당 Filter에 대한 정의를 구현한 Method를 생성하는 방법입니다.

1. Mozart Explorer에서 Simluation/Filters 노드를 선택하고 Add Item 버튼을 누릅니다.





2. 새로 작성할 Filter Method 명을 입력합니다. Factor Name은 Filter Manager에 등록한 Filter 이름과 동일하게 설정해야 합니다.
3. 생성된 Method는 Filters Category에 자동으로 추가가 됩니다.

4. Method를 정보 입력이 완료되면 솔루션 탐색기의 Logic 메뉴 하단에 Filters 클래스를 생성합니다. 이미 존재하는 경우 Method만 등록하며, 기존에 Filters 클래스가 없는 경우에만 생성합니다.
5. Method 원형을 자동이 자동 생성되면 Method를 편집할 수 있도록 해당 Method가 포함된 .cs 파일을 소스편집기에 열리고, 커서가 해당 함수 위치합니다. 함수의 Return value 는 boolean이며 True 가 반환이 되면 해당 작업물언 Filtering이 되어 Dispatching 대상에서 제외됩니다.

Simulation 통계 수집(Statistics)

Mozart Simulation 라이브러리에는 제조업 시뮬레이션의 목적을 고려하여 다음의 기본 통계정보를 제공하며, 사용자가 일부 Option을 조정하도록 지원합니다.

- 공정의 생산실적(Move) 통계: 기간별 공정,장비의 생산실적을 집계하여 제공합니다.
- 재공(Wip)통계: 일정 주기마다 시점별 재공현황 정보를 집계하여 제공합니다.
- 장비 로딩이력과 가동율 통계: 장비에서의 작업물 로딩 이력정보와 기간별 장비의 상태별 점유비율 정보를 제공합니다.

공정 생산실적 통계 사용방법

재공 통계 사용방법

장비 통계 사용방법

사용자 정의 통계 작성 방법

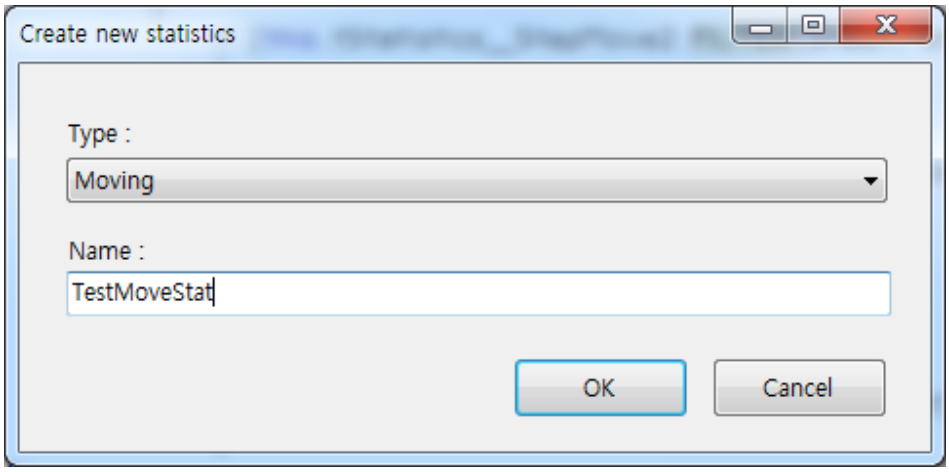
Move Statistics

시뮬레이션 수행 결과로서의 공정 실적 통계량을 집계하기 위해 사용합니다. Entity 기반의 통계 집계 방식을 사용하되 공정투입 실적과 공정생산(Out) 실적을 집계할 수 있도록 구성되어 있습니다. 공정의 In/Out 실적은 Track-In, Track-Out 시점에 집계합니다. Move 실적도 바라보는 관점이 장비입장인지 공정입장인지에 따라 다른 Key 를 지정할 수 있습니다.

시스템에서 제공하는 Move Type Statistics 를 추가하는 방법에 대해 알아봅니다

통계량 추가

Statistics>AddItem 메뉴를 사용하여 Type 을 `Moving` 으로 설정하여 처리합니다. 아래그림과 같이 Type 을 선택하고 명칭을 입력한 후 `OK` 버튼을 누르면 Tree 에 해당 명칭의 노드가 생기고 Output 및 입력항목을 설정할 수 있는 입력창이 열립니다.



Output Schema 설정

기본적인 사용자 정의 통계를 정의하는 것과 같이 최종 Output 데이터가 저장된 Output data item 을 선택합니다. 이때, 해당 Data item 에 Key 가 설정되어 있어야 합니다. 해당 Key 는 데이터를 찾을때 사용되는 Key 로 Unique value를 찾을 수 있는 Key 로 정의합니다.

컬럼의 Collect Type 설정

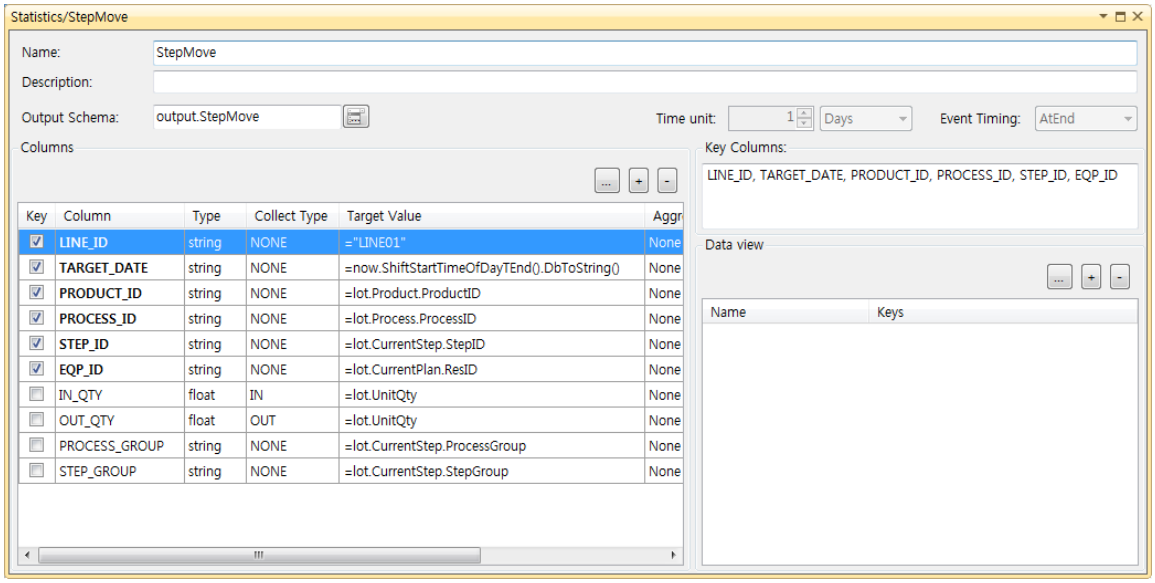
Move Type 통계량 설정시에는 Collect Type = {"NONE", "IN", "OUT"} 이 제공됩니다. 아래는 항목별 처리 방식입니다.

- NONE : 별도 처리를 하지 않습니다.
- IN : 작업물의 Track-In 시점에 설정된 Target Value 를 기준으로 수량을 추가하여 집계합니다. TrackIn 시점에 처리되어야 할 기준 Rule 이 포함되어 있는 것입니다.
- OUT : 작업물의 Track-Out 시점에 설정된 Target Value 를 기준으로 수량을 추가하여 집계합니다.

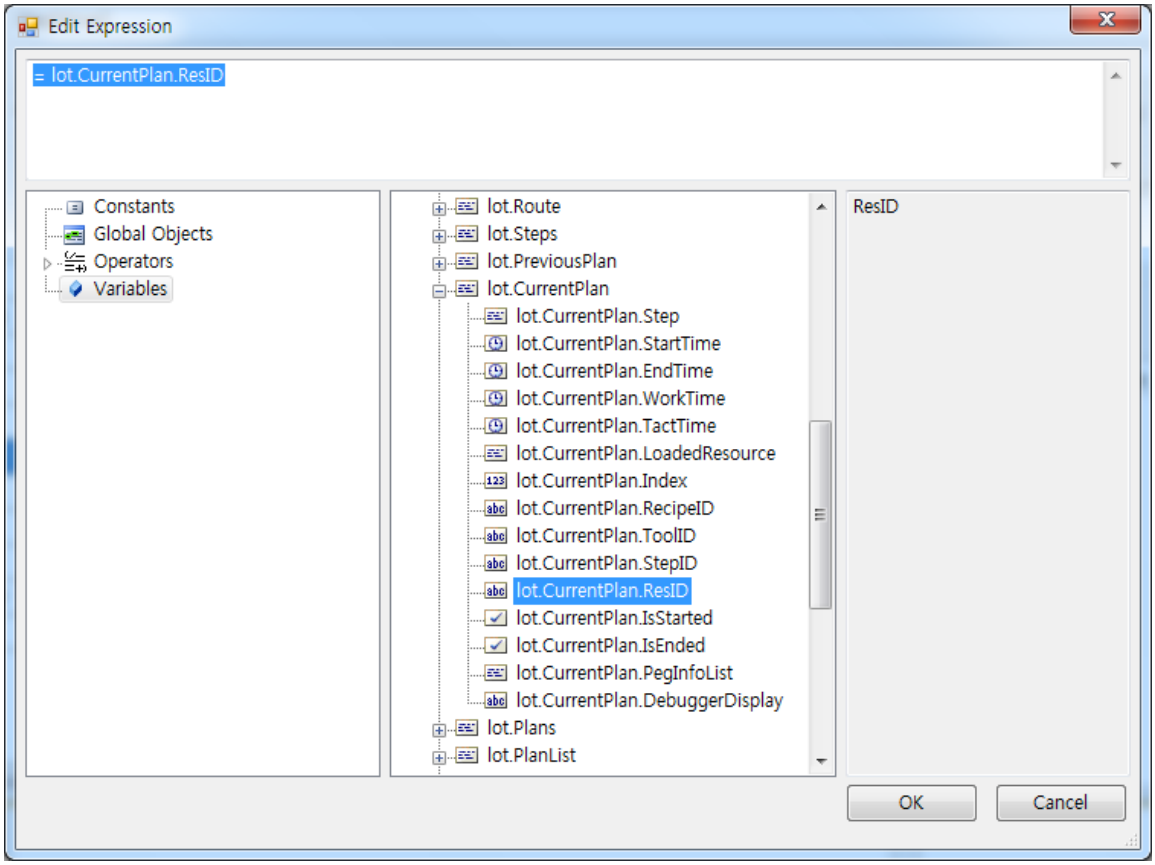
1. Target Value 설정

   Target Value 는 Output 데이터의 컬럼과 집계시점에 사용할 수 있는 변수의 Mapping 정보입니다. 간단한 Expression 의 사용이 가능합니다. 아래 그림은 같이 Step, Eqp 단위의 실적을 집계한다고 했을 때 Target Value를 정의한 그림입니다. Key 에 해당하는 컬럼의 경우 Target Value는 작업물과 작업물가공 장비와 같은 속성을 Mapping 하며, 집계 결과의 경우 Collect Type 을 설정하고, Unit 단위 혹은 Lot 단위의 추가 수량을 지정합니다.

   단, 실적 집계 단위를 나타내는 속성의 경우 Move 량을 집계하는 시간 주기를 잘 설정합니다. 아래 예에서는 시간 단위 실적을 집계하도록 설정한 것을 볼 수 있습니다.



   변수설정 및 Expression 사용시 사용할 수 있는 사전정의된 변수(Variables)는 Target Value 편집을 위한 Expression 편집기에 표시됩니다. 라이브러리의 사용이 익숙한 경우 Expression 편집부분에 직접 코드를 작성할수도 있습니다.



2. 기본 통계량 수집시 Rolling 주기 및 Event Type 고정

   상기와 같은 설정만으로 통계량의 집계가 가능합니다. Move Type 통계량의 경우 데이터 기록 주기는 1일로 고정되고 Rolling Event 는 동시간대 가장 마지막에 발생하도록 고정됩니다.

3. Moving type 통계량 집계시의 커스터마이징 방식

   1,2,3 과정을 통해 기본 Move 기반통계량의 집계가 가능합니다. 다만, 사용자가 Move 집계시 추가적인 정보를 가공하여 저장하고자 하는 경우, 혹은 자동 생성되는 코드와 다른 처리를 하려는 경우 통계 아이템 하부에 생성되는 Event Handler 를 구현하여 커스터마이징 할 수 있습니다. 아래는 Event Handler 별 호출 시점 및 사용용도입니다.

   - InitializeStat : 통계 데이터의 초기화를 위해 필요합니다. 예를 들어 시뮬레이션 결과로 특정 기간 전체의 실적을 기록해야 하는데 시뮬레이션이 기간 중 시작되는 경우 시뮬레이션 시작 이전의 실적을 본 함수에서 미리 만들어두는 작업을 수행할 수 있습니다.

   - InitializeRow : 하단의 GetRow 함수 혹은 Key 에 의해 변경대상 Row 정보를 찾아 없는 경우 새로운 데이터를 생성하게 되며 이때, Collect Type = IN, OUT 을 제외한 속성들을 설정하기 위한 함수입니다. 기본적으로는 Target Value 에 설정한 값으로 속성을 할당하지만 사용자가 지정한 방식으로 처리하기 위해서는 본 함수를 구현합니다.

- Filter : 통계 집계시 작업물 혹은 작업물의 로딩 정보상 실적 집계시 제외해야 하는 데이터가 있다면 본 함수를 통해 해당 데이터를 Filtering 할 수 있습니다. 예를 들어 장비로딩 실적만 집계한다고 했을 때 작업물의 로딩 장비 정보가 없는 경우 Filter 에서 이를 판별하여 **"true"** 를 반환하면 데이터가 집계에서 제외됩니다.
- GetRow : 작업물의 TrackIn, TrackOut 시점에 동일 Key 를 가진 Row 를 찾아주는 코드가 자동 생성됩니다. 이때 설정된 Key 정보를 사용합니다. 그러나 기존 Output 에서 동일 데이터를 찾는 방식을 변경하고자 하는 경우에는 본 함수를 구현합니다. 기존 데이터에서 현재 작업물의 Move 결과를 기록하기 위한 동일 Row 를 찾는 로직을 구현합니다.
- OnTrackIn : 작업물 Monitor 에서 작업물이 TrackIn 하는 시점에 자동생성된 코드를 처리한 이후에 호출됩니다. 사용자는 작업물 TrackIn 시점에 추가적인 정보를 가공할 수 있습니다.
- OnTrackOut : 작업물 Monitor 에서 작업물이 TrackOut 하는 시점에 자동생성된 코드를 처리한 이후에 호출됩니다. 사용자는 작업물 TrackOut 시점에 추가적인 정보를 가공 할 수 있습니다.

Wip Statistics

Wip 타입의 통계 데이터는 특정 시점의 재공 수량 정보를 집계하기 위한 기능입니다. Wip 데이터는 Move 정보와 다르게 집계기간동안의 누적수치가 아닌 특정시점의 상태별 재공수량 정보를 집계하도록 설계되었습니다. 따라서 작업물의 상태가 변경되는 시점에서 모두 데이터를 집계할 수 있도록 Design 되어 있습니다. 해당 시점은 아래와 같습니다. 작업물의 상태는 EntityState 로 정의되며 {"HOLD", "MOVE", "WAIT", "RUN", "OUT_WAIT" } 의 다섯가지 상태로 구분됩니다. 따라서 이러한 이벤트 발생시 작업물의 상태는 아래와 같이 변경됩니다.

- TransferIn : "MOVE" 상태 재공 증가
- TransferOut : "MOVE" 상태 재공 감소
- HoldIn : "HOLD" 상태 재공 증가
- HoldOut : "HOLD" 상태 재공 감소
- DispatchIn : "WAIT" 상태 재공 증가
- DispatchOut : "WAIT" 상태 재공 감소
- CancelWait : "WAIT" 상태 재공 감소

- TrackIn : "RUN" 상태 재공 증가
- TrackOut : "RUN" 상태 재공 감소
- OutWaitIn : "OUT_WAIT" 상태 재공 증가
- OutWaitOut : "OUT_WAIT" 상태 재공 감소

Wip Type Statistics 는 상기 상태변경 시점에 Key 에 따른 재공의 가감을 자동으로 처리하도록 되어 있어 사용자는 Output 에 대한 속성을 지정하는 것으로 통계량을 수집할 수 있습니다. 아래는 Wip Type 통계량을 추가하는 방법입니다.

---

0. Wip Type Statistics 추가

Statistics>AddItem 메뉴를 사용하여 Type 을 "Wip" 으로 설정하여 처리합니다. "Create new statistics" 다이얼로그에서 Type 을 선택하고 명칭을 입력한 후 "OK" 버튼을 누르면 Tree 에 해당 명칭의 노드가 생기고 Output 및 입력항목을 설정할 수 있는 입력창이 열립니다.

---

1. Output Schema 설정

기본적인 사용자 정의 통계를 정의하는 것과 같이 최종 Output 데이터가 저장된 Output data item 을 선택합니다. 이때, 해당 Data item 에 Key 가 설정되어 있어야 합니다. **해당 Key 는 데이터를 찾을때 사용되는 Key** 로 Unique value를 찾을 수 있는 Key 로 정의합니다.

---

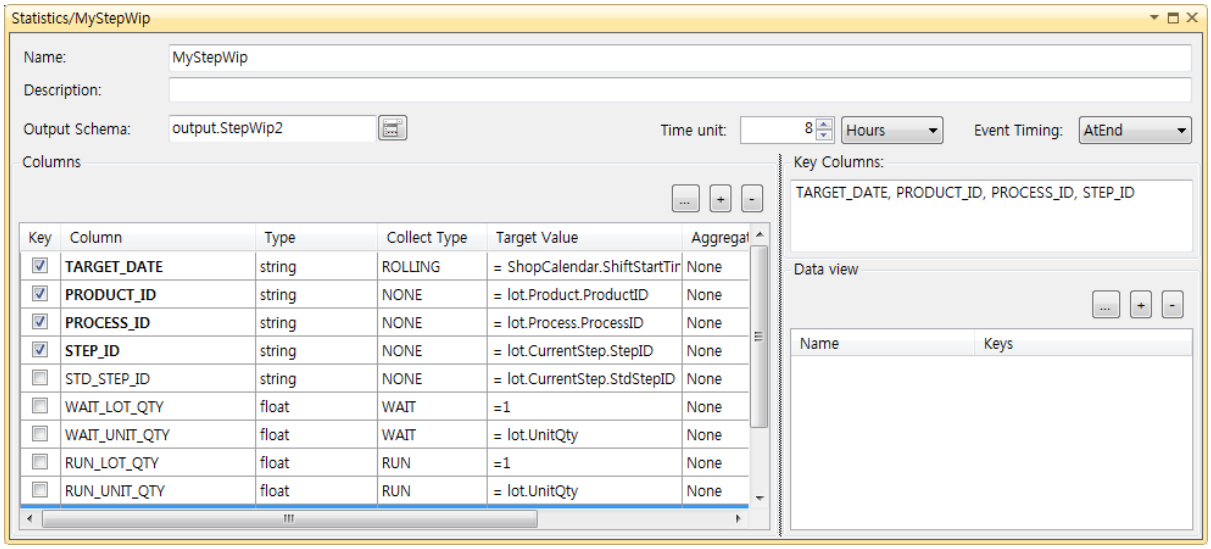
2. 컬럼의 Collect Type 설정

Move Type 통계량 설정시에는 아래의 타입이 제공됩니다. 아래는 항목별 처리 방식입니다.

- NONE : 별도 처리를 하지 않습니다.
- HOLD : HoldIn, HoldOut 이벤트에 대해 자동으로 설정된 수량을 가감하도록 처리합니다.
- MOVE : TransferIn, TransferOut 이벤트에 대해 자동으로 설정된 수량을 가감하도록 처리합니다.
- WAIT : DispatchIn, DispatchOut, CancelWait 이벤트에 대해 자동으로 설정된 수량을 가감하도록 처리합니다.

- RUN : TrackIn, TrackOut 이벤트에 대해 자동으로 설정된 수량을 가감하도록 처리합니다.
- OUT_WAIT : OutWaitIn, OutWaitOut 이벤트에 대해 자동으로 설정된 수량을 가감하도록 처리합니다.
- ROLLING : Wip 의 Time Key 에 해당하는 속성에 설정합니다. Rolling 으로 설정된 속성은 Time unit 으로 설정된 주기에 따라 기존 Record 의 해당 속성항목이 Target Value 에 설정한 값으로 업데이트됩니다. 이과정을 통해 이전 시점의 EOH 가 다음 시점 BOH 가 되고, 다음 Rolling 주기가 되기전까지 동일 데이터의 상태별 재공 수량을 업데이트하게 됩니다.

아래 그림의 경우 WAIT, RUN WIP 에 대한 집계만 하도록 설정한 모습이며 8시간 단위로 Rolling 하도록 처리하여 Shift 별 EOH 기준의 재공을 집계하도록 설정한 예시입니다. ROLLING 타입으로 설정한 속성의 Target Value 와 상단의 Time unit 은 반드시 동일한 주기로 설정되어야 합니다. 그렇지 않은 경우에는 데이터가 중접되는 경우가 발생할 수 있습니다. 아래 예의 경우 TARGET_DATE 를 SHIFT 시작시각으로 설정 (ShopCalendar.ShiftStartTime(now)하고 Time unit 을 8시간으로 설정하여 주기를 맞춘 것을 확인 할 수 있습니다.



3. Target Value 설정

Target Value 는 컬럼의 Collect Type 과 Key 를 고려하여 사전 정의된 변수를 사용하여 정의합니다. Expression 을 사용할 수 있도록 편집창을 제공하며, 작업물과 관련된 변수를 제공합니다.

4. Rolling 주기 및 Event Timing 설정

Time unit 은 Output data item 에 데이터를 기록하고 통계량 집계 데이터를 삭제하는 주기를 설정하는 것입니다. **Moving 타입**과 다르게 Wip 타입 통계량은 이를 설정할 수 있으며, 2번 항목에서 설명한 것과 같이 Output 컬럼의 Collect Type 이 "ROLLING" 으로 설정된 항목의 Target Value 와 주기를 맞추어 주어야 합니다. 만일 1시간 단위의 Wip 을 기록해야 한다면 Time unit = 1 hour 로 설정하고 "ROLLING" 타입의 컬럼에 대한 Target Value = now.ToString("yyyyMMdd HH0000") 과 같이 시간단위로 값을 설정하는 방식입니다.

Event Timing 의 경우 기본값은 AtEnd 입니다. 이 옵션은 Rolling Event 와 동일 시점에 발생하는 Event 가 있는 경우에 Rolling 을 가장 마지막에 수행하도록 처리합니다. 반대로 동일 시점 Event 중 가장 먼저 실행려면 AtBegin 을 설정합니다. 통상 통계치의 경우 동일 시점에 발생하는 다른 Event 의 영향을 받는 경우가 많음으로 AtEnd 로 기본 설정되는 것입니다.

5. Wip type 통계량 설계시 커스터마이징 방식

Wip 통계량 추가시 생성되는 Event Handler 의 경우 기본적인 항목은 Moving 타입과 같은 기능을 담당합니다. 아래 4가지 항목입니다.

- InitializeStat : 통계 데이터의 초기화를 위해 필요합니다. 예를 들어 시뮬레이션 결과로 특정 기간 전체의 실적을 기록해야 하는데 시뮬레이션이 기간 중 시작되는 경우 시뮬레이션 시작 이전의 실적을 본 함수에서 미리 만들어두는 작업을 수행할 수 있습니다.
- InitializeRow : 하단의 GetRow 함수 혹은 Key 에 의해 변경대상 Row 정보를 찾아 없는 경우 새로운 데이터를 생성하게 됩니다. 기본적으로는 Target Value 에 설정한 값으로 속성을 할당하지만 사용자가 지정한 방식으로 처리하기 위해서는 본 함수를 구현합니다.
- Filter : 재공 집계시 작업물 속성상 재공 집계시 제외해야 하는 데이터가 있다면 본 함수를 통해 해당 데이터를 Filtering 할 수 있습니다. "true" 를 반환하면 데이터 집계에서 제외됩니다.
- GetRow : 작업물의 TrackIn, TrackOut 시점에 동일 Key 를 가진 Row 를 찾아주는 코드가 자동 생성됩니다. 이때 설정된 Key 정보를 사용합니다. 그러나 기존 Output 에서 동일 데이터를 찾는 방식을 변경하고자 하는 경우에는 본 함수를 구현합니다. 기존 데이터에서 현재 작업물의 Move 결과를 기록하기 위한 동일 Row 를 찾는 로직을 구현합니다.

추가적으로 Wip 의 상태 변경 시점 직후에 사용자가 추가적으로 통계량 수집을 처리할 수 있도록 각 Event 직후 호출되는 Event Handler 를 정의할 수 있습니다. 각 이벤트에 "On" 접두어를 붙여 생성됩니다. 예를 들어 TrackIn 처리를 한 후 호출되는 Handler 는 "OnTrackIn", DispatchOut 후에 호출되는 Handler 는 "OnDispatchOut" 과 같은 형태입니다. 기본적인 통계로 처리가능한 경우에는 해당 Handler 에 함수를 정의하지 않습니다.

Eqp Statistics

장비 통계는 장비의 상태가 변경되는 시점에 통계량을 집계하는 방식입니다. Mozart 의 장비 상태는 LoadingState= {"SETUP","BUSY", "IDLERUN", "IDLE", "PM", "DOWN", "WAIT_SETUP"} 로 정의되며 크게 아래 3가지 유형의 통계집계기능을 제공합니다.

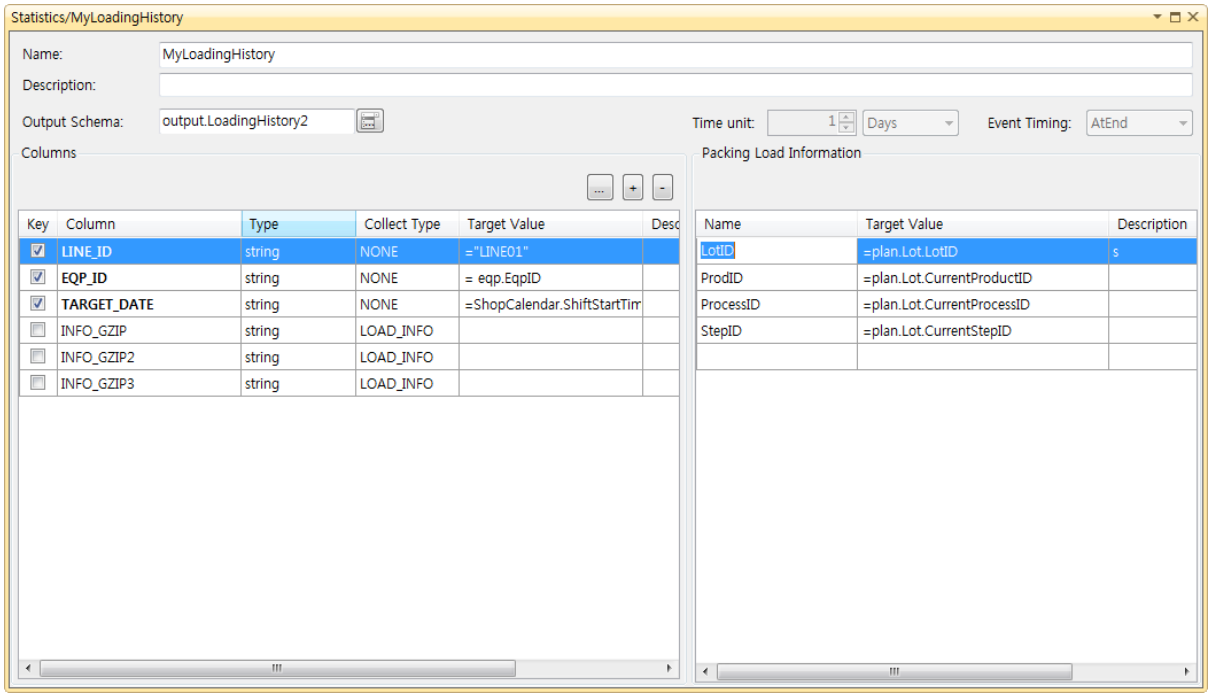
1. 장비 상태 변경 이력

2. 장비 상태별 점유비율(가동율)

3. 장비별 작업물 로딩 이력(계획)

Equipment Loading History

장비의 상태 변경이력을 기록한 기본적인 장비 통계입니다. 상태변경은 일반적으로 작업물의 로딩과 함께 이루어 지는 경우가 많음으로 설정화면에는 "BUSY" 상태로 변경되는 시점의 로딩된 작업물 정보를 설정할 수 있도록 되어 있습니다. 아래그림은 장비 통계 중 LoadingHistory 를 추출하기 위한 Statistics 설정 화면입니다. 추가시에는 "LoadHist" 타입을 선택하여 추가합니다.



설정순서

1.Output Schema 설정

최종 Output 을 기록할 Output data item 을 선택합니다. 이때 해당 Output 에 Key가 사전에 설정되어 있어야 합니다. Key 가 설정되지 않은 경우에 GetRow 함수를 구현해 주어야 합니다.

2.컬럼별 Collect Type 지정

Collect Type 은 "LOAD_INFO" 타입 하나만 존재합니다. LOAD_INFO 는 기본적으로 아래와 같은 형식으로 정의됩니다.

형식 : 상태;시작시각;{Packing Load Information} {TAB} 상태;시작시각;{Packing Load Information} {TAB} ....

예시 : BUSY;142540;L004;PROD01;PROC01;S200
BUSY;182540;L205;PROD01;PROC01;S200

3.컬럼별 Target Value 설정

Collect Type 이 없는 경우에 컬럼별로 Mapping 되는 Target Value 를 Expression 과 사전정의된 변수를 사용하여 정의합니다.

4.Packing Load Information 설정

상기 2번에서 LOAD_INFO 로 정의된 데이터에 장비가 "BUSY"인 경우 로딩된 작업물의 정보를 표시하게 됩니다. 이때 로딩 정보에 포함될 정보를 정의합니다. 설정된 항목별로 ";" 을 구분자로 기록됩니다. 상기 이미지의 예시에서는 LotID;ProductID;ProcessID;StepID 로 Packing 데이터를 구성한 모습을 볼 수 있습니다. 만일 로딩시 사용한 Recipe 의 정보를 함께 기록하거나 PlanInfo 에 추가적으로 정의한 속성정보를 같이 표시하고자 하는 경우 본 설정 화면에서 정의 할 수 있습니다.

5.Rolling 주기 및 Event Timing 고정

LoadHist 타입의 경우 데이터는 하루 단위로 Rolling 되어 Output data 에 기록되며, 메모리에서 삭제됩니다. 또한 Rolling Event 는 동일시점 Event 에서 가장 마지막에 실행되도록 설정되며 변경이 불가합니다.

Model Reference

- Output Data Schema Template : **LoadingHistory schema**

LoadingHistory Schema

Schema template category : Outputs

Schema template name : LoadingHistory

Schema description : 장비의 상태 변경 이력을 기록한 Output

LoadingHistory Schema

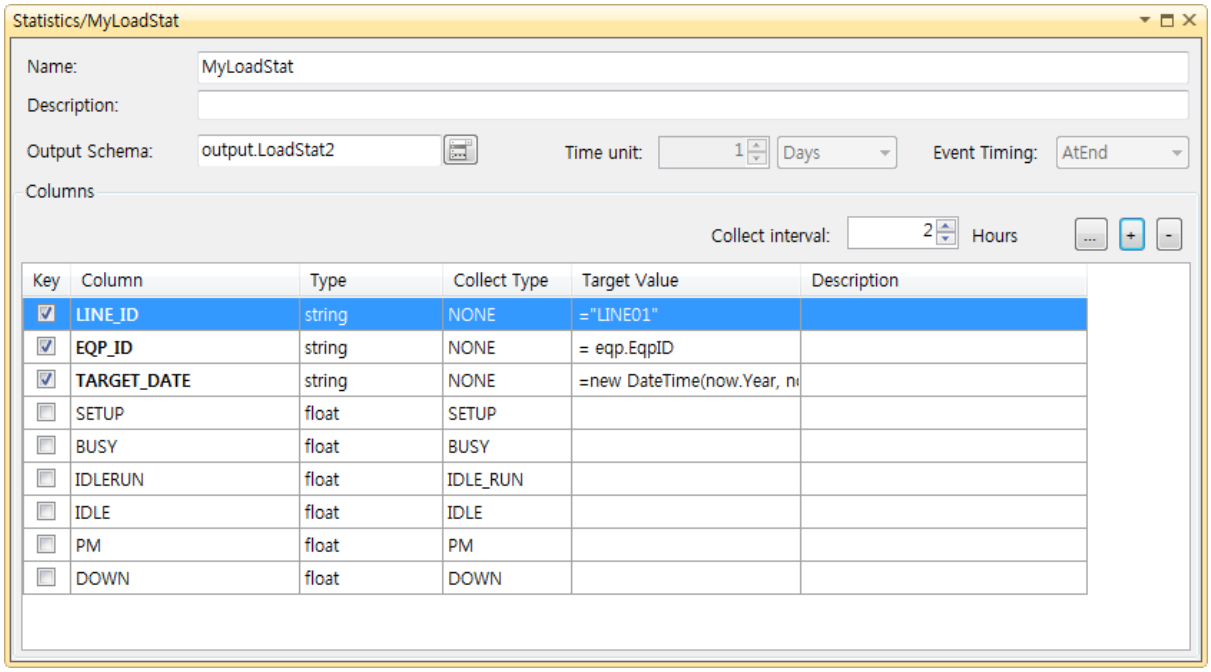
Aa Column Name	Data Type	Description	Modeling Guide

Column Name	Data Type	Description	Modeling Guide
VERSION_NO	string	계획버전 번호	계획 수립 시점별로 생성되는 고유 번호
LINE_ID	string	라인 ID	장비의 소속 라인 ID
EQP_ID	string	장비 ID	로딩 상태변화 기록 대상 장비 ID
TARGET_DATE	string	생산집계 시점	YYYYMMDD 형태의 문자열입니다. 통상 장비 상태 변화를 일별로 데이터를 압축해서 기록하게 됨
INFO_GZIP	string	상태변경 기록 저장 문자열 1	상태변경 정보는 데이터이 커 상태와 변경시점 및 간단한 당시 로딩 정보를 기록하게 되며, 문자열의 길이가 긴 경우 문자열2, 3에 나누어 기록할 수 있습니다. 이후 결과를 사용할때에닌 기록당시 BUSY 상태인 경우 로딩 정보를 기록하는 방식에 따라 문자열을 Parsing 하여 사용해야 합니다.
INFO_GZIP2	string	상태변경 기록 저장 문자열 2	상태변경 기록 저장 문자열 2
INFO_GZIP3	string	상태변경 기록 저장 문자열 3	상태변경 기록 저장 문자열 3

Schema template file : LoadingHistory.vditem

Equipment Utilization Statistics

장비의 상태별 점유비율, 점유시간을 기록하는 통계정보입니다. Mozart 에서 장비의 상태 (LoadingState) 는 {"SETUP", "BUSY", "IDLERUN", "IDLE", "PM", "DOWN", "WAIT_SETUP" } 의 7가지 상태로 관리되며, 설정에 따라 각 상태/수집주기별 점유비율에 대한 통계자료를 수집할 수 있습니다. 아래는 상태별 통계를 설정하는 화면입니다. 추가시에는 "LoadStat" Type 을 선택하여 추가합니다.



1.Output Schema 설정

최종 Output 을 기록할 Output data item 을 선택합니다. 이때 해당 Output 에 Key가 사전에 설정되어 있어야 합니다. Key 가 설정되지 않은 경우에 GetRow 함수를 구현해 주어야 합니다.

2.컬럼별 Collect Type 지정

Collect Type 은 LoadingState 의 각 상태타입을 선택할 수 있도록 되어있습니다. 다만, WAIT_SETUP 상태는 SETUP 상태로 집계하도록 합니다. Collect Type 이 "NONE" 이 아닌 경우 해당 상태의 점유비율 데이터가 해당 컬럼에 설정됩니다.

3.컬럼별 Target Value 설정

Collect Type 이 있는 경우 해당 컬럼 값은 자동으로 설정된 Collect Type 의 상태 점유비율이 설정됩니다. 만일 그렇지 않은 경우에는 Target Value 에 설정된 값이 할당됩니다. 이중 상태 점유 비율의 ***수집 주기(기간)에 해당하는 속성이 필요하며 이는 Collect interval 과 주기가 맞아야 합니다.***

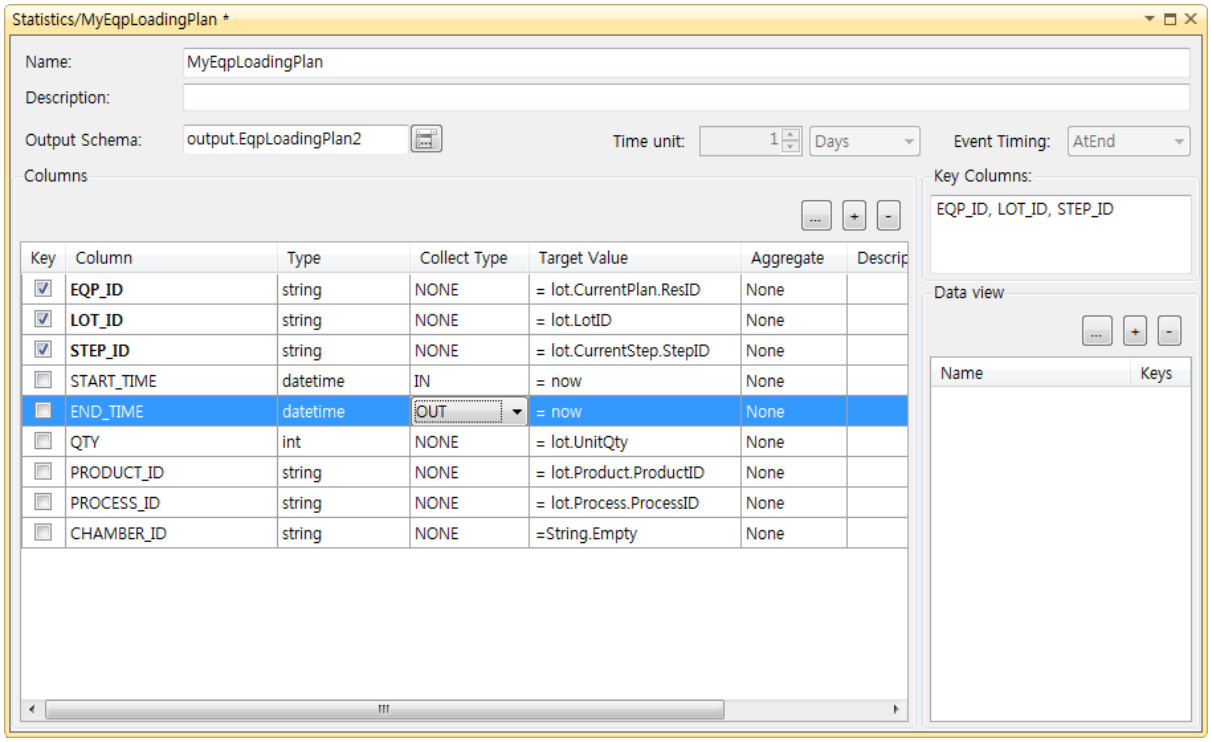
예를 들어 Collect interval 을 2시간으로 설정한 경우(상기 예)에는 Time Key 에 해당하는 컬럼의 값도 2시간 단위 값을 설정해주어야 합니다. 상기 예에서는 "da.NowDT.Date.AddHours(da.NowDT.Hour - (da.NowDT.Hour % 2)).DbToString()" 와 같은 방식으로 2시간 단위 Time Key 를 설정하도록 처리합니다.

4.Rolling 주기 및 Event Timing 고정

EqpPlan 타입의 경우 데이터는 하루 단위로 Rolling 되어 Output data 에 기록되며, 메모리에서 삭제됩니다. 또한 Rolling Event 는 동일시점 Event 에서 가장 마지막에 실행되도록 설정되며 변경이 불가합니다.

Eqp Plan

장비에 로딩된 작업물의 이력정보입니다. 작업물의 Track In, Track Out 을 기준으로 작성되도록 구성되어 있습니다. 아래 그림은 EqpPlan 통계를 출력하기 위한 설정 예시입니다. 통계 추가시에는 "EqpPlan" 을 선택합니다. 장비 통계이긴 하나 작업물의 상태 변경 Event 를 기준으로 작성하도록 구성되어 있습니다. Moving 통계와 구성은 거의 같습니다.



1.Output Schema 설정

최종 Output 을 기록할 Output data item 을 선택합니다. 이때 해당 Output 에 Key가 사전에 설정되어 있어야 합니다. Key 가 설정되지 않은 경우에 GetRow 함수를 구현해 주어야 합니다.

2.컬럼별 Collect Type 지정

Collect Type 은 "IN", "OUT" 타입 중에 선택하며, 통상 TrackIn 시점의 시각, TrackOut 시점의 시각을 기록하며 그외에는 Target Value 로 속성을 설정하도록 합니다. 상기 예시를 참조합니다.

3.컬럼별 Target Value 설정

Collect Type 이 없는 경우에 컬럼별로 Mapping 되는 Target Value 를 Expression 과 사전정의된 변수를 사용하여 정의합니다. 기본적으로 제공되는 변수는 lot, now, hb 세가지 유형입니다. lot 은 Project 에서 정의된 단위 작업물 객체를 참조할 수 있으며 hb 를 통해 배치구성시 작업물 정보를 참조할 수 있습니다. now 변수를 통해 현재 시각을 참조할 수 있습니다.

4.Rolling 주기 및 Event Timing 고정

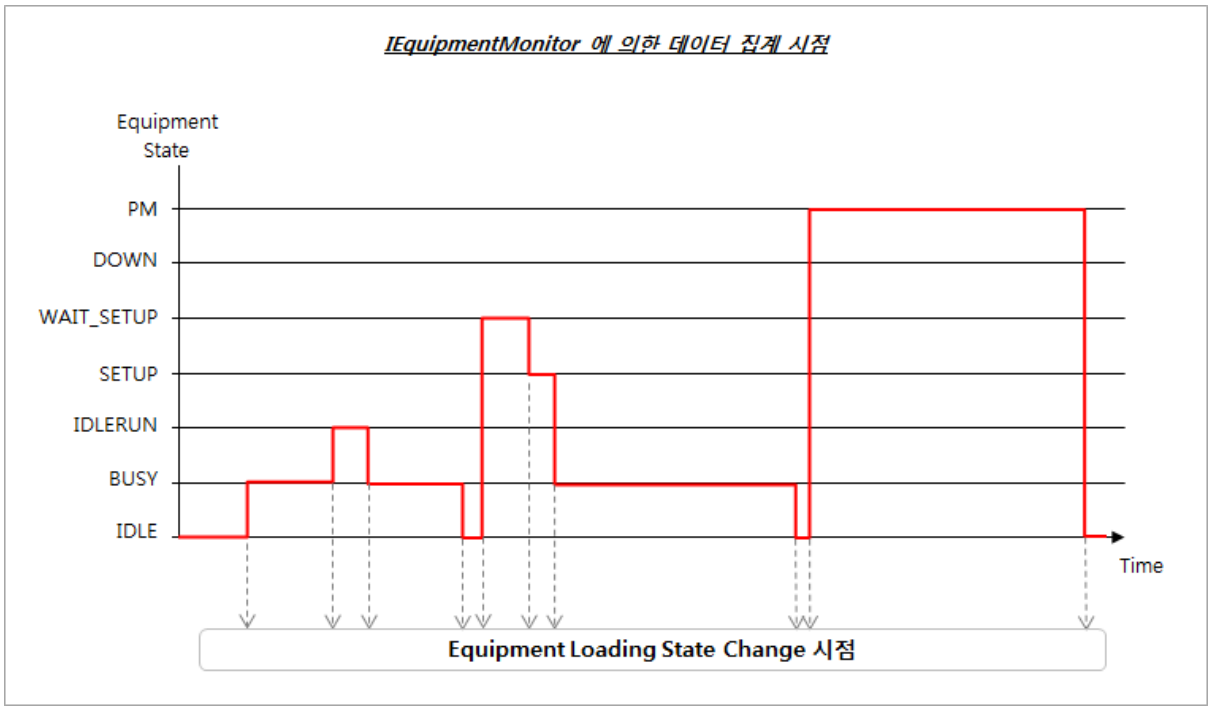
EqpPlan 타입의 경우 데이터는 하루 단위로 Rolling 되어 Output data 에 기록되며, 메모리에서 삭제됩니다. 또한 Rolling Event 는 동일시점 Event 에서 가장 마지막에 실행되도록 설정되며 변경이 불가합니다.

사용자 정의 통계 수집

시뮬레이션 결과 통계 수집을 위한 기본적인 두가지 방식은 장비의 상태를 기반으로 통계를 수집하는 방식과 작업물의 상태변경 시점에 통계량을 수집하는 방식이 있습니다. 아래는 Mozart 에서 두가지 관점의 상태와 상태변경 시점과 통계수집 방식을 설명합니다.

장비 상태 기반 통계 수집 개념

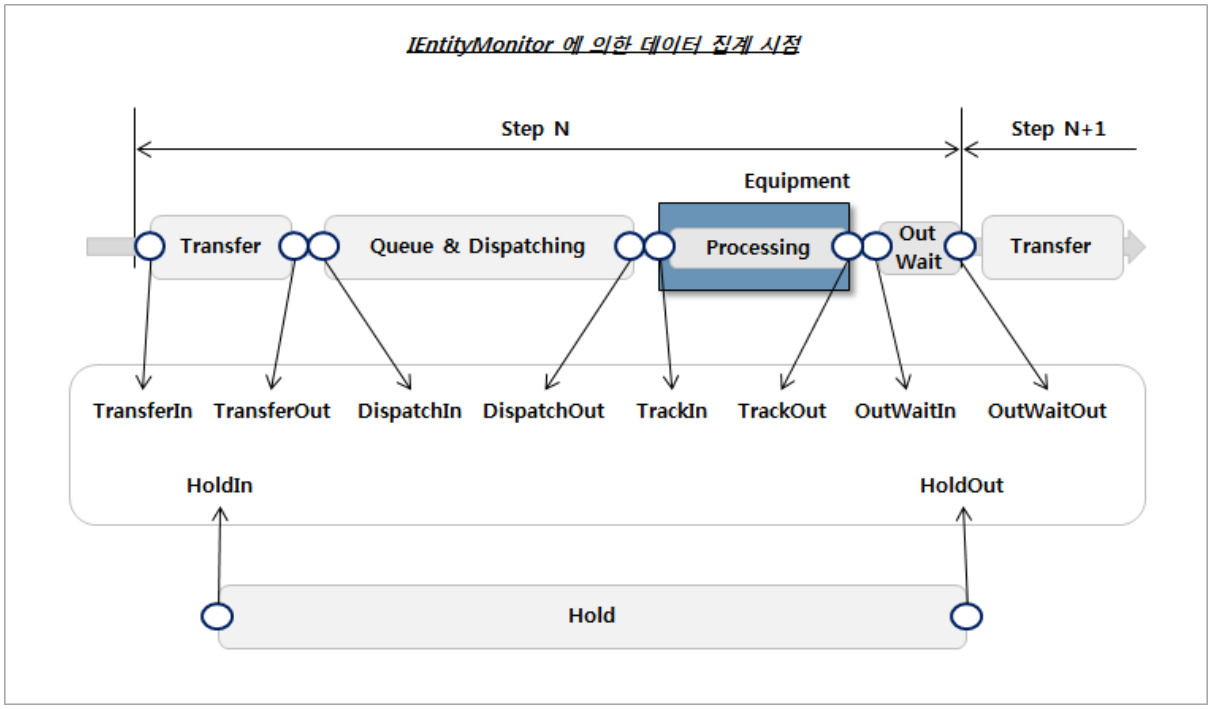
아래 그림과 같이 장비는 작업물의 로딩/언로딩, PM, 고장 등의 Event 에 따라 상태가 전이됩니다. 장비에 대한 통계는 장비의 상태가 변경되는 시점에 수집데이터의 형태에 따라 결과를 가공하여 저장하는 방식으로 집계합니다. 예를 들어 장비의 상태가 변경되는 이력을 기록한다면 아래 그림의 상태 변경시점을 기록하면 됩니다. 또한 일정기간동의 각 상태별 점유비율을 집계하는 경우 기간의 시작시점부터 상태별 점유시간을 기록하는 방식으로 이를 기록할 수 있습니다. Mozart IDE의 Simulation Module 에서는 이러한 장비통계를 보다 쉽게 기록할 수 있도록 Statistics type 을 제공합니다. 그러나 사용자가 보다 복잡하고 다양한 Output 을 집계하고자 하는 경우에는 Custom Type 의 Statistics 를 생성하여 필요한 Event 에 통계치를 생성하도록 작업할 수 있습니다.



장비의 상태 변경 Event 처리시에는 Custom statistics 하단에 생성되는 Event handler 중 "StateChanged" 를 구현하여 처리합니다.

작업물 기반 통계 수집 개념

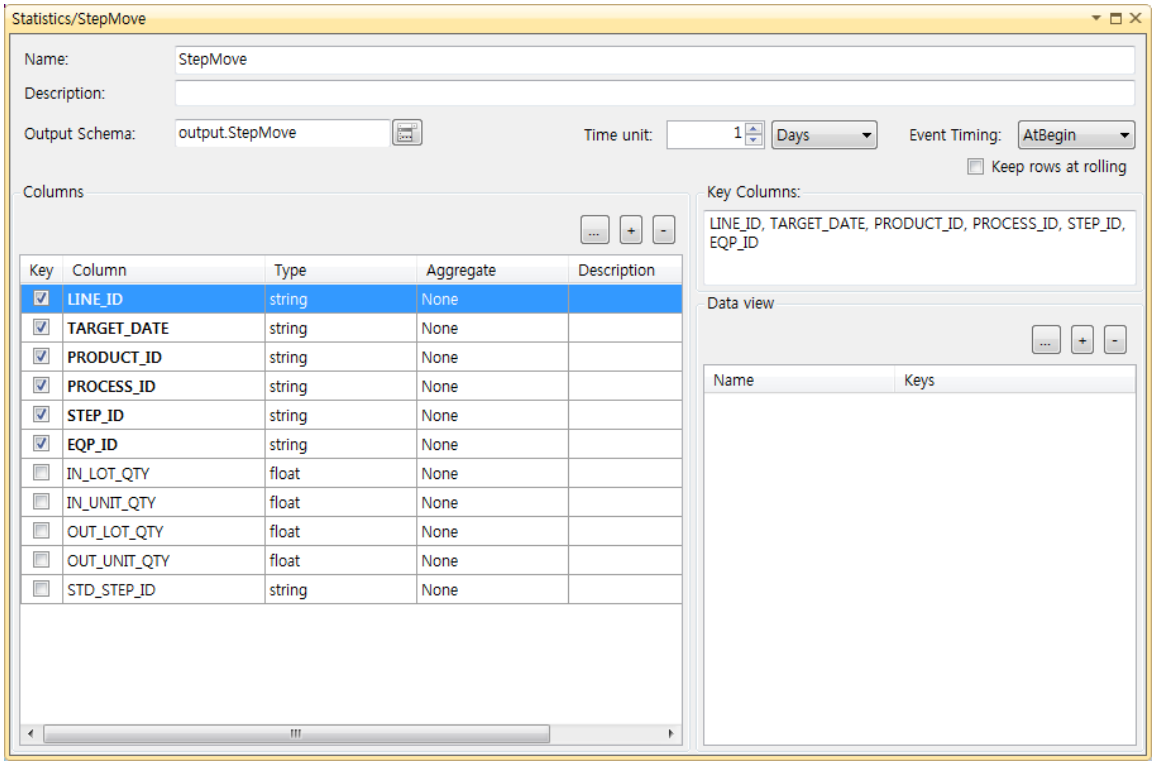
작업물 기반 통계는 작업물의 상태가 변경되는 지점에서 수집하게 되며, Mozart 에서는 이러한 작업물 기반 통계를 쉽게 수집할 수 있도록 Statistics 기능을 제공합니다. 아래 그림은 공정 이동에 따른 작업물의 상태변화 시점을 개념적으로 도출한 것으로 Statistics 기능을 통해 통계를 정의할 때 하기 지점에서 필요한 작업을 할 수 있도록 기능이 구성되어 있습니다.



작업물 기반 통계기능으로 Step 의 실적(Move) 량 집계, Step TAT의 집계 및 재공의 집계, 특정 공정(Fab Out, Fab In)의 Moving 량 집계 등의 다양한 통계 수치를 집계할 수 있습니다.

Custom Statistics 작성방법

1. **Simulation/Statistics** 노드에서 Popup 메뉴로 **Add Item** 메뉴를 클릭합니다.
2. Type 을 "Custom" 으로 선택하고 수집데이터 명을 입력합니다.
3. 입력한 명칭으로 Statistics 하위로 통계치가 생성되고, 입력창이 활성화됩니다. 각 입력값을 입력합니다.



- Name : 통계 명칭
- Description : 통계 설명
- Output Schema : 콤보박스에서 Model/Output 에 등록한 Schema 리스트 중 대상 Schema 를 선택합니다. 만일 신규 Schema 가 필요한 경우에는 Output DataItem 을 추가한 후 다시 작업을 수행해야 합니다.
- Time Unit : 메모리상에 수집된 통계치를 Reset 하는 주기로 Reset 직전에 데이터를 Output File 에 저장하게 됩니다. 또한 주기에 따라 Aggregate 컬럼의 통계치를 업데이트합니다.
- Event Timing : Time Unit 의 주기에 따라 발생하는 Rolling event 의 실행순서를 정합니다.
  - AtBegin : 동일 시점에 발생한 Event 들 중 가장 먼저 실행합니다.
  - AtEnd : 동일 시점에 발생한 Event 들 중 가장 나중에 실행합니다.
- Keep rows at rolling : Rolling 이 발생할때 기본적으로는 설정된 Output schema 의 data item 에 결과를 복사하고 메모리에서 데이터를 삭제하나 본 옵션이 체크된 상태에서는 삭제하지 않습니다.
- DataView : Columns 에 정의된 Schema 기준의 데이터 집계 테이블을 사용하여 로직에서 필요한 데이터 뷰 컬럼의 Key 를 지정합니다. Data view 그리그 상단의**[+][-]** 버튼을 이용해서 View 의 Key 를 지정합니다.

- Columns : Output Schema 에서 선택한 데이터의 컬럼이 표시되며, 컬럼을 추가할 수 있습니다. 물론 추가한 컬럼은 Output 에 표시되지는 않으며, 시뮬레이션 중 사용자 코드에서 참조는 가능합니다.

> **Note**
> 컬럼의 속성을 설정할 때 다음과 같은 작업을 수행할 수 있습니다. Key 컬럼의 경우 Output Schema 에 설정된 Data Item 의 Key 를 따르며, 수정을 불가합니다.

  - Type 설정 : 속성 데이터 타입을 설정합니다.
  - Aggregate 속성 설정 : 특정 컬럼의 경우 데이터 Key 를 기준으로 시간에 따른 값의 변화에 대한 Min, Max, 평균 등의 통계치로 설정을 할 수 있습니다. 이 경우 소스상에서 해당 컬럼값을 설정할때 StatSheetCfg 의 Update 함수를 사용하여 처리합니다. Primary Key 컬럼은 Aggregate 속성을 지정할 수 없습니다.

4. 기본 통계 속성을 입력한 후 모델 탐색기의 해당 통계아이템 노드를 펼치면 작업물 통계 수집 시점별 Event 처리 Point 가 표시됩니다.

Event 설명

```
공통 이벤트

- InitializeStat : 통계 데이터의 초기화를 위해 필요합니다. 예를 들어 시뮬레이션 결과로 특정 기간 전체의 실적을 기록해야 하는데 시뮬레이션이 기간 중 시작되는 경우 시뮬레이션 시작 이전의 실적을 본 함수에서 미리 만들어두는 작업을 수행할 수 있습니다.
- OnRolling : 설정한 Rolling 별로 호출됩니다. 통계량 수집에 주기 별로 추가적으로 필요한 작업을 수행 할 수 있습니다.
- OnWriteOutput : Rolling 시점에 통계량 정보를 사용자가 알아서 판단하여 기록합니다.  이 이벤트가 활성화 되어 있지 않은 경우, Rolling 주기로 결과가 기록됩니다.
- OnWrap : 해당 Row의 Aggregate 컬럼 값을 설정한 후 호출되어 집니다. 설정된 Aggregate 컬럼 값을 확인하실 수 있습니다.
- InitialzeRow : 하단의 GetRow 함수 혹은 Key 에 의해 변경대상 Row 정보를 찾아 없는 경우 새로운 데이터를 생성하게 되며 이때, Collect Type = IN, OUT 을 제외한 속성들을 설정하기 위한 함수입니다. 기본적으로는 Target Value 에 설정한 값으로 속성을 할당하지만 사용자가 지정한 방식으로 처리하기 위해서는 본 함수를 구현합니다.

작업물 통계 수집 이벤트

- TransferIn/Out : 작업물이 이동을 시작하는 시점, 이동 종료되는 시점에 호출되는 이벤트입니다.
- HoldIn/Out : 작업물이 Hold 상태로 변경되는 시점과 Hold 가 풀리는 시점에 호출되는 이벤트입니다.
- DispatchIn/Out : 작업물이 공정 대기를 시작하고, 대기후 선택되어 대기 종료하는 이벤트입니다.
- CancelWait : 작업물이 DispatchIn 후 대기를 시작하다가 대기취소 되는 경우 발생하는 이벤트입니다.
- TrackIn/Out : 작업물이 장비에 TrackIn, Out 되는 시점에 호출되는 이벤트입니다.
- OutWaitIn/Out : 작업물이 장비의 작업을 마치고 다음 공정으로 이동하기 전 대기하는 경우 대기 시작/완료시점에 호출되는 이벤트입니다.

장비 통계 수집 이벤트
```

```
- StateChanged : 장비의 상태 변경시점에 호출되는 이벤트입니다.
```

1. 통계치 수집에 필요한 Event 를 선택하고 Popup 의 Edit 메뉴를 클릭하면 해당 Event 처리 함수를 구현할 수 있도록 소스편집창이 활성화됩니다.
2. StatSheetCfg 및 StatSheet 클래스의 기능을 활용하여 통계 수집코드를 작성합니다. (클래스 레퍼런스 참조)

Move 집계 예시

TrackIn, TrackOut Event 에서 작업물의 In, Out Move 실적을 집계하는 예제입니다.

```
public void TRACKIN(StatSheet<StepMove> sheet, ISimEntity entity)
{
    if (entity is SiteLotBatch)
    {
        UpdateStepMove(sheet, (entity as SiteLotBatch).Contents, true);
    }
    else
    {
        UpdateStepMove(sheet, new List<ISimEntity>() { entity }, true);
    }
}
public void TRACKOUT(StatSheet<StepMove> sheet, ISimEntity entity)
{
    if (entity is SiteLotBatch)
    {
        UpdateStepMove(sheet, (entity as SiteLotBatch).Contents, false);
    }
    else
    {
        UpdateStepMove(sheet, new List<ISimEntity>() { entity }, false);
    }
}
private void UpdateStepMove(StatSheet<StepMove> sheet, IList<ISimEntity> entities, bool
 isTrackIn)
{
    foreach (SiteLot lot in entities)
    {
        StepMove target = GetStepMove(sheet, lot);
        if (isTrackIn)
        {
            target.IN_LOT_QTY++;
            target.IN_UNIT_QTY += lot.NumOfUnits;
        }
        else
        {
            target.OUT_LOT_QTY++;
            target.OUT_UNIT_QTY += lot.NumOfUnits;
        }
    }
}
```

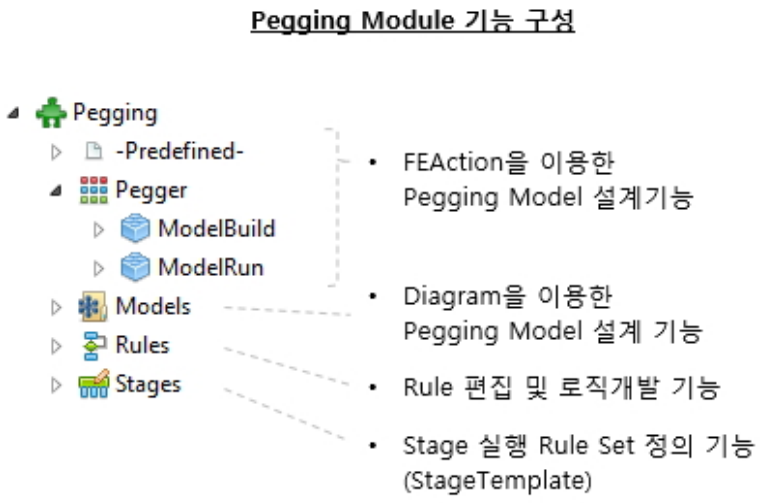
```
private StepMove GetStepMove(StatSheet<StepMove> sheet, SiteLot lot)
{
    string line = lot.LineID;
    string product = lot.Product.ProductID;
    string process = lot.Process.ProcessID;
    string step = lot.CurrentStep.StepID;
    string eqp = string.Empty;
    if (lot.CurrentPlan.LoadedResource == null)
    {
        eqp = "-";
    }
    else
    {
        eqp = lot.CurrentPlan.LoadedResource.ResID;
    }
    string targetDate = DateUtility.DbToString(ShopCalendar.ShiftStartTimeOfDayEndT(shee
t.NowDT));
    StepMove item = sheet.GetRow(line, targetDate, product, process, step, eqp);
    return item;
}
```

PEGGING MODULE

Pegging 모듈 개요

MOZART Engine 은 기본적으로 Planning & Schedule 수행을 위한 1단계로, 공정별 Target 생성하고 Target 을 충족하기 위한 Plan & Schedule 결과를 생성하도록 구성되어 있습니다. Target 생성을 위한 기본 Logic 은 Demand 를 Backward 전개하여 재공을 Pegging 하고 잔량에 대한 공정별 In/Out Target 생성하는 방식입니다.

Pegging Module은 Pegging Logic 개발을 위해 MOZART SeePlan Pegging Library 와 Pegging FEComponent 를 제공합니다. SeePlan Pegging Library 는 MOZART Rule Flow Engine 을 기반으로 Pegging Logic 개발에 적합한 형태의 도구집합을 제공합니다. 기능은 크게 Pegging Model 구성과 실행제어를 위한 기능, Pegging Logic 을 쉽고 유연하게 개발 할 수 있도록 Rule, Stage 를 생성 관리할 수 있는 세 가지 기능으로 구분됩니다.



- **FEAction 을 이용한 Pegging Model 설계기능** : Mozart SeePlan, Semicon Library 의 Pegging FEComponent 를 통해 FEAction을 노출하고 구현 Logic (FEDefinition) 을 관리할 수 있습니다. 자세한 사항은 **Pegging FEModel** 을 참고하시기 바랍니다.
- **Diagram을 이용한 Pegging Model 설계기능** : Pegging Logic 개발 도구 중 Diagram 을 그리는 방식으로 Pegging 수행에 필요한 Model 을 작성하는 기능을 제공합니다.
- **Pegging Rule 편집기능** : Pegging 의 Logic 실행 단위인 Rule 을 신규로 등록/수정/삭제 할 수 있는 기능을 제공합니다.
- **Pegging Stage 편집기능** : Rule Flow Engine 에서 Rule Set 을 처리하는 단위인 Stage 를 신규로 등록/수정/삭제할 수 있는 기능을 제공합니다.

Rule 편집

Rule 은 Stage 에 포함되며, Stage 실행 시에 실행되는 Logic 의 최소 실행단위입니다. SeePlan Pegging Library 에서는 Pegging 에서 사용되는 기본 Rule 은 제공을 하며, 그 외에 추가적인 Rule 이 필요한 경우 사용자가 Rule 편집 기능을 통해 Rule 추가하거나 수정/삭제 할 수 있습니다.

Rule 편집 방법

1. MOZART Explorer 의 **[Pegging > Rules]** 노드를 선택합니다.
2. 팝업 메뉴에서 **[Add Item]** 을 실행합니다.
3. Pegging Rule 생성 다이얼로그 에서 Rule 명 과 Rule 에 대한 설명을 입력하고 **[OK]** 버튼을 클릭합니다.
4. Rule 의 소스코드 편집창이 활성화되면 Rule 의 수행 Logic 을 입력합니다.

> 💡 **Note**
> Rule 의 입력 Parameter 와 Return 값은 모두 StageTargetGroup 으로 StageTargetGroup 은 개발 사이트에서 상속받아 재정의한 Data 클래스로 타입캐스팅하여 사용합니다. 제조업 Pegging Project 의 경우 StageTargetGroup 은 생산제품별 Demand 정보가 됩니다.

Rule Implementation Sample Code

```
// 수율을 반영하는 Rule 샘플
public IStageTargetGroup ApplyYield(IStageTargetGroup target)
{
    PlanPart pp = target as PlanPart ;
    float yield = GetYield(pp.CurrentStage);
    foreach (MoStep ms in pp.TargetList)
    {
        ms.CalcQty = ms.CalcQty / yield;
    }
    return target;
}
```

Stage 편집

Stage 는 Rule Flow Engine 의 Model 을 구성하는 주요 요소로서, Logic 실행과 전체 Flow 구성의 기본 단위가 됩니다. Stage 간 연관 관계에 대한 Model 은 Main/Control FEComponent 를 통해 사용자가 재정의 할 수 있으며, Stage 편집 기능을 통해 Stage 유형을

정의하고 유형별로 실행해야하는 Rule 및 Rule 간 선후관계를 정의할 수 있습니다. 자세한 사항은 **Step Stage 의 Rule 구성방안**을 참고하시기 바랍니다.

Stage 편집 방법

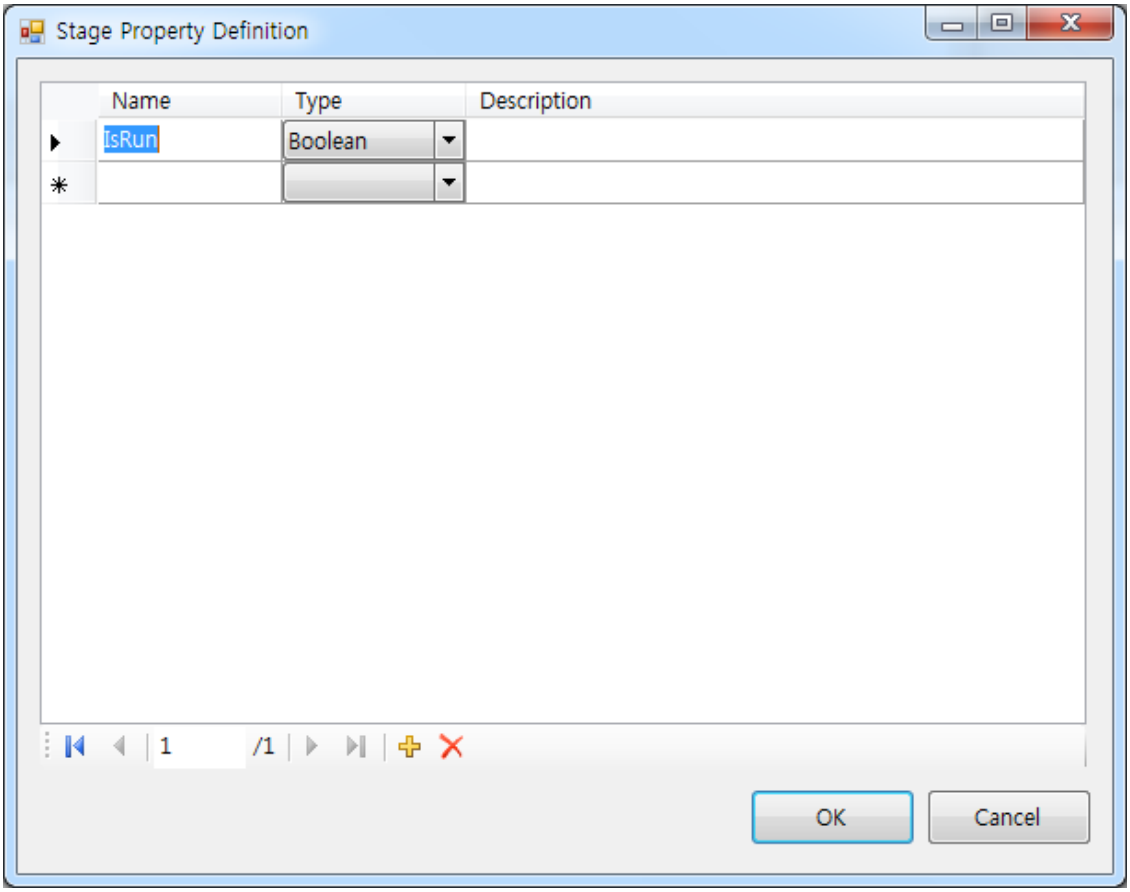
1. MOZART Explorer 의 **Pegging > Stages** 노드를 선택합니다.
2. 팝업 메뉴의 **[Add Item]** 명령을 실행합니다.
3. Stage 명을 입력합니다.
4. Stage 편집 창이 활성화되면 Stage 에 대한 설명을 입력하고, **"Rule Call Sequence"** List 의 상단 **[+]**버튼을 사용하여 Rule List 를 조회 하고 Stage 에서 사용할 Rule 을 선택하여 추가합니다.

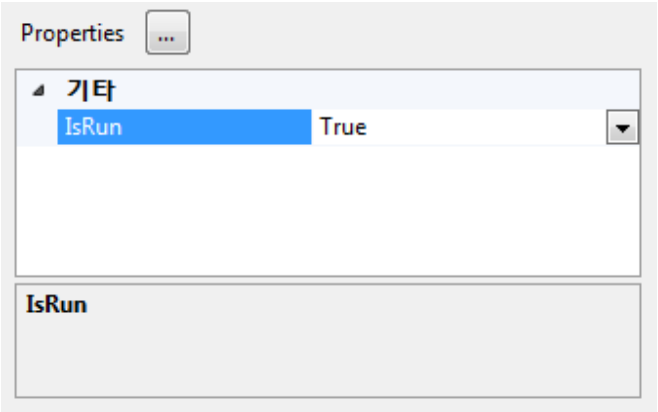
   > 💡 **Note**
   > Pegging Rule List 에는 기본적으로 참조된 SeePlan Library 와 Domain Library 에 정의된 Rule 과 사용자가 Project에서 신규로 정의한 모든 Rule 이 표시됩니다.

5. Stage 에서 실행되어야 할 모든 Rule 의 선택이 완료되면 상단의 ⬆️ ⬇️ 버튼을 통해 Rule의 순서를 조정할 수 있습니다.
6. Stage 의 모든 Rule 에 대한 편집이 완료되면 입력창을 저장합니다.
7. 저장된 결과로 MOZART Explorer 의 Stages 하위에 생성한 Stage 가 추가되며, 해당 Stage 노드를 펼치면 Stage 에서 실행될 Rule 이 순서대로 표시됩니다.

   > 💡 **Note**
   > 사용자가 Project 에서 추가한 Rule (버튼이름) 은 Stage 하위의 Rule Node 를 통해 바로 소스편집이 가능합니다. 다만, Library 에서 제공되는 기본 Rule (버튼이름) 의 경우에는 편집이 불가합니다.

8. Stage 의 Run, Wait 상태를 정의하기 위해서는 Stage 편집창의 Properties 항목에 Property 를 설정해야 합니다. 설정하기 위해서는 Properties 옆의 **[...]** 버튼을 눌러 Property 를 정의합니다. 기본 Property 인 Run, Wait Phase 속성은 기본 추가되어야 합니다. 속성명은 "IsRun" 으로 설정합니다





속성을 추가한 후 Run Phase 의 Stage 인 경우 True 를, Wait Phase 의 경우 False 를 설정합니다.

Pegging 컨트롤 라이브러리

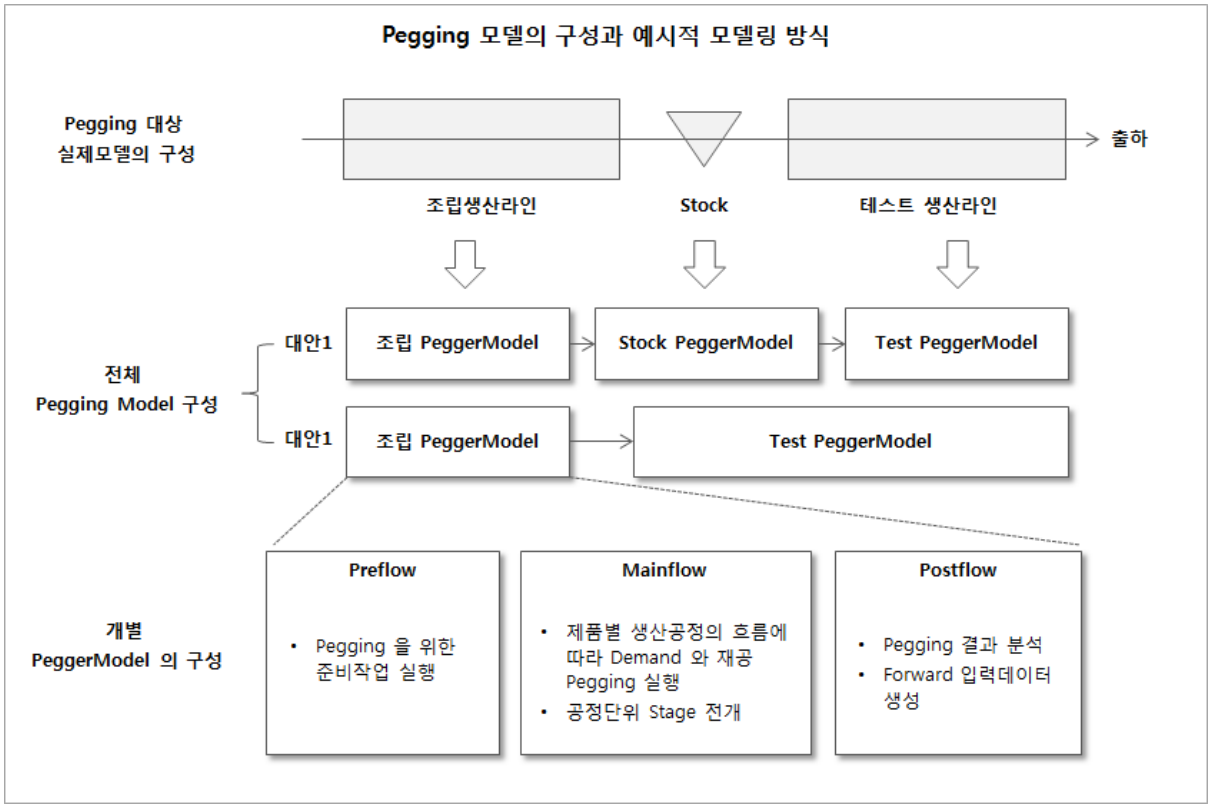
개요

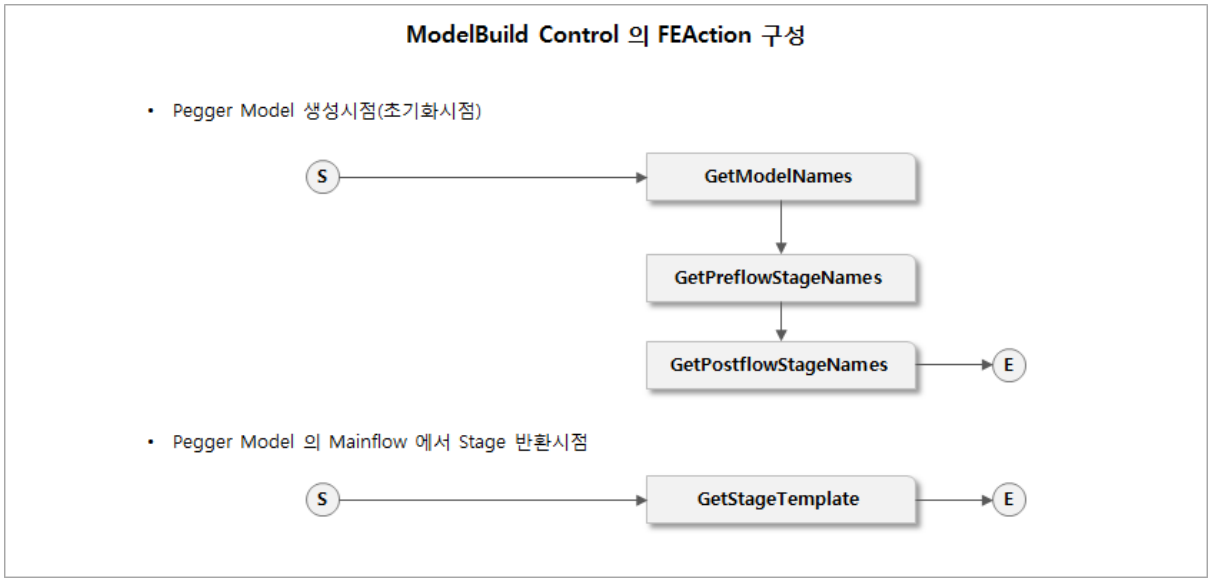
Pegging FEModel 은 SeePlan Pegging Library 의 Core Model 을 재정의 할 수 있도록 아래의 두 가지 FEComponent 로 구성됩니다.

- **ModelBuild FEComponent**
  사이트의 특성에 맞는 Pegging Model 을 구성하기 위한 FEAction 의 집합입니다.
- **ModelRun FEComponent**
  Pegging Model 의 실행 시 개별 Rule 외에 Pegging Model 의 실행을 제어하기 위해 필요한 FEAction 의 집합니다.
- **Rules**
  Pegging 실행 시 일반적으로 사용되는 기본 Logic 을 제공합니다. 사용자는 사전 정의된 Rule 에서 노출한 FEAction 을 통해 Logic 의 Customizing 을 할 수 있습니다.

ModelBuild Control

Pegging 수행을 위한 Pegging Model 구성을 정의하기 위한 FEAction 의 집합입니다. 아래 그림과 같이 실제 제조라인의 구성에 따라 Pegging 을 어떤 방식으로 수행할지 결정(모델링)하고 결과를 시스템에 반영하도록 Logic 을 구현할 수 있습니다.





1. ***GetModelNames*** : 전체 Pegging Model 을 구성하는 PeggerModel 이름 List 를 반환하는 코드를 작성합니다. 대상 제조시스템을 분석하고 PeggerModel 구성방안을 정의한 후에 구현할 수 있습니다. 예를 들어 반도체 Fab 의 경우, 라인 내에 명시적이거나 개념적인 Stock 을 두고 Pegging 을 나누어서 할 필요성이 없으므로 하나의 PeggerModel 로 전체 Pegging 을 처리할 수 있습니다. 그러나, Stock 및 재공이 일정시간 혹은 조건에 따라 저장되었다가 다음 공정으로 흘러가는 구조의 경우, 필요에 따라 Model 을 분리해야 할 수 있으며 이러한 경우 복수 개의 Model 을 구성할 수 있습니다. 구현 예는 아래와 같습니다.

구현 예시

```
public List<string> GETMODELNAMES (ref bool handled, List<string> preReturnValue)
{
   // 반도체 Fab 과 같이 단일 Model 을 구성하는 경우
   return new List<string>() { "Fab" };
}
public List<string> GETMODELNAMES (ref bool handled, List<string> prevreturnValue)
{
   // 두 개의 분리된 라인 Group 과 중간에 Stock 을 두고 있는 제조라인의 Pegging Model 을 생성하는
경우
   // Model 순서대로 실행순서 결정

   return new List<string>() { "Area", "Stock", "Area2" };
}
```

2. ***GetPreflowStageNames*** : 상기 지정한 Model 의 Preflow 를 구성하는 Stage 명을 실행순서를 고려하여 List 로 반환합니다. 구현 예는 아래와 같습니다.

   구현 예시

```
public string[] GETPREFLOWSTAGENAMES (string modelName, ref bool handled, string[] p
revReturnValue)
{
   // 입력 값에 해당하는 Model 별로 실행대상이 되는 Stage 명을 반환하도록 코드 구성
   // 반도체 Fab 과 같이 단일 Model 로 구성하는 경우에는 modelName 에 따라 변경할 필요가 없음
   return new string[] { "InitTargets", "InitWips" };
}
```

3. ***GetPostflowStageNames*** : Model 별 Postflow 를 구성하는 Stage 명을 실행순서를 고려하여 List 로 반환합니다. Stage 명을 반환하는 코드 작성 시 사전에 해당 Stage 가 등록되어 있어야 정상적으로 Logic 이 작동합니다. Stage 를 통해 해당 Stage 의 실행 시 호출될 Rule 과 Rule 간 실행순서를 정의합니다.

4. ***GetStageTemplate*** : 각 Model 의 Main Flow 의 경우, Preflow, Postflow 와 같이 지정된 Stage 의 순서로 실행되는 구조가 아닌 제품에 따른 Process 별 공정순서에 의해 진행되며, 해당 공정별 Pegging Rule 의 경우 일반적으로 동일한 구성을 따릅니다. 그러나 동일 공정의 상태 (Run, Wait) 에 따라 Pegging 을 달리해야 하는 경우 혹은 여러 번의 Pegging 을 해야하는 경우가 있습니다.

> 💡 **Note**
> **e.g.)**Run Pegging, Wait Pegging 시 기본 Product ID 가 동일한 Demand 에 우선 Pegging 하고 재공이 남는 경우에 해당 Product 와 대표 제품이 같은 제품의 Demand 에 2차적으로 Pegging 을 할 수 있습니다.

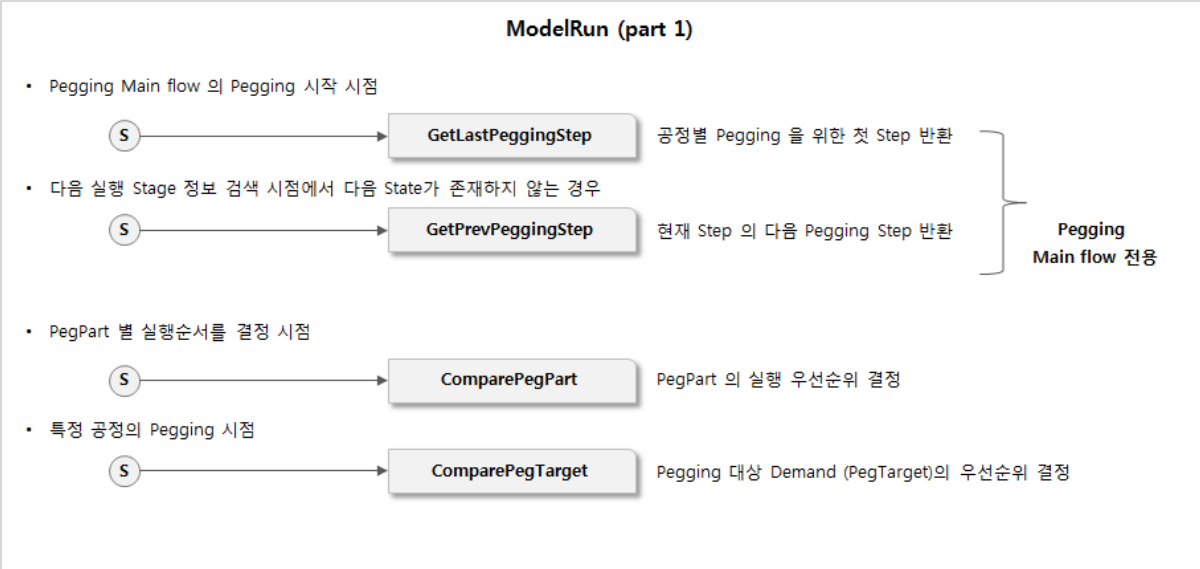
이러한 경우, 한 Step 을 복수의 Stage 로 처리할 수 있으므로 본 Action 에서는 Step 을 몇 개의 Stage 로 구성할지 여부를 결정하고 해당하는 Stage 명을 순차적으로 반환합니다. 구현 예는 아래와 같습니다.

구현 예시

```
// 아래의 코드를 작성하기 전에 해당 Stage 명에 대한 Stage 를 생성해야 함
public string[] GETSTAGETEMPLATE (Step step, ref bool handled, string [] prevReturnV
alue)
{
   return new string[] { "Run1", "Run2", "Run3", "Wait1", "Wait2", "Wait3" };
}
```

ModelRun Control

Pegging Model 의 실행 시 개별 Rule 외에 Pegging Model 의 실행을 제어하기 위해 필요한 FEAction 의 집합니다. 본 컨트롤의 경우 Pegger Model 의 Main flow 의 Stage 순서를 제어하는 부분과 PegPart, PegTarget 에 대한 Pegging 우선순위, PegPart (Demand) 에 대한 Merge, Split 을 제어하기 위한 부분으로 구성됩니다.



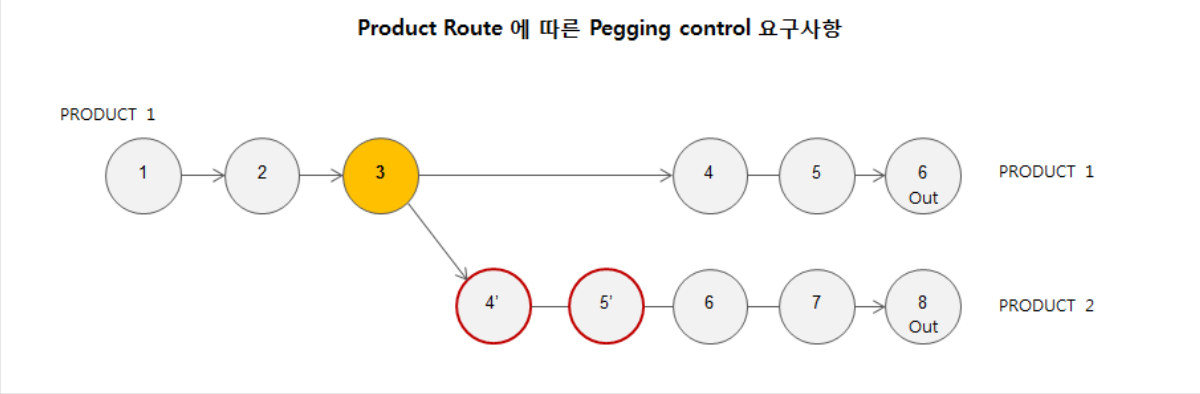
- **GetLastPeggingStep** : Pegging Model 실행 시 Target (Demand) 에 대한 제품별 첫 번째로 수행할 Pegging Step 을 찾아 반환합니다. Pegging 은 기본적으로 Flow (Process) 를 구성하는를 마지막 Step 이 첫 번째 Pegging Step 이 됨으로 Action 명은

이를 반영하였으며, SemiFAB Library 에 포함된 기본 Logic 은 이를 반영하도록 Logic 이 구성되어 있습니다. 공정별 Pegging 을 수행하는 Main flow 의 경우에는 제품별로 해당 Stage 를 찾는 Logic 을 별도로 구현할 수 있습니다.

구현 예시

```
public virtual Step GET_SEMI_LAST_PEGGING_STEP(PegPart pegPart, ref bool handled, Step p
revReturnValue)
{
    // Process 의 가장 마지막 Step 부터 Backward 로 Stage 를 만들었으므로
    // Flow 의 첫 step 을 반환하면 됨
    SemiconPegPart pp = (SemiconPegPart)pegPart;
    SemiconProcess proc = (SemiconProcess)pp.Product.Process;
    SemiconStep first = (SemiconStep)proc.LastStep;
    return first;
}
```

- **GetPrevPeggingStep** : 현재 Pegging 중이 Step 의 다음 실행 Pegging Step 정보를 찾아 반환합니다. Pegger Model 을 구성하는 Pre, Post Flow 의 경우 일반적으로 ModelBuild 시 Stage 의 선후관계를 설정함으로 기본 Logic 에 의해 Logic 이 수행되며, 공정별 Pegging 을 수행하는 Main flow 의 경우에 본 Action 의 결과 Step 에 대한 Stage 가 실행됩니다. 반도체 Fab 과 같이 Process 를 구성하는 공정이 많은 경우에는 조건에 따라 특정 공정에 대한 Pegging 을 Skip 하는 등의 업무 Logic 을 본 Action을 통해 구현할 수 있습니다. Pegging 은 Backward 로 진행됨에 따라 Flow 상 이전 Step 이 다음 실행대상 Pegging Step 이 됩니다.



- **ComparePegPart** : Backward 로 공정을 전개하며 PegPart 별로 Pegging 을 수행할 때 특정 시점에 PegPart 의 Current Stage 기준으로 Pegging 수행의 우선순의를 PegPart 별로 지정할 필요가 있는 경우 사용합니다. 예를 들어 Forward 기준으로 특정 공정에서 Product 가 분기되는 경우에 Backward 기준으로는 해당 공정에서 두 개의 PegPart 에 대한 Demand 가 통합되어야 합니다. 그러나 Out 공정을 기준으로 해당 분기되는 공정까지의 공정수가 다른 경우 분기되는 공정에서는 모든 분기되는 Product에 대한 PegPart 의 Current Stage 가 동일 Step 에 오기전까지 해당 Step 의 Pegging 을 수행

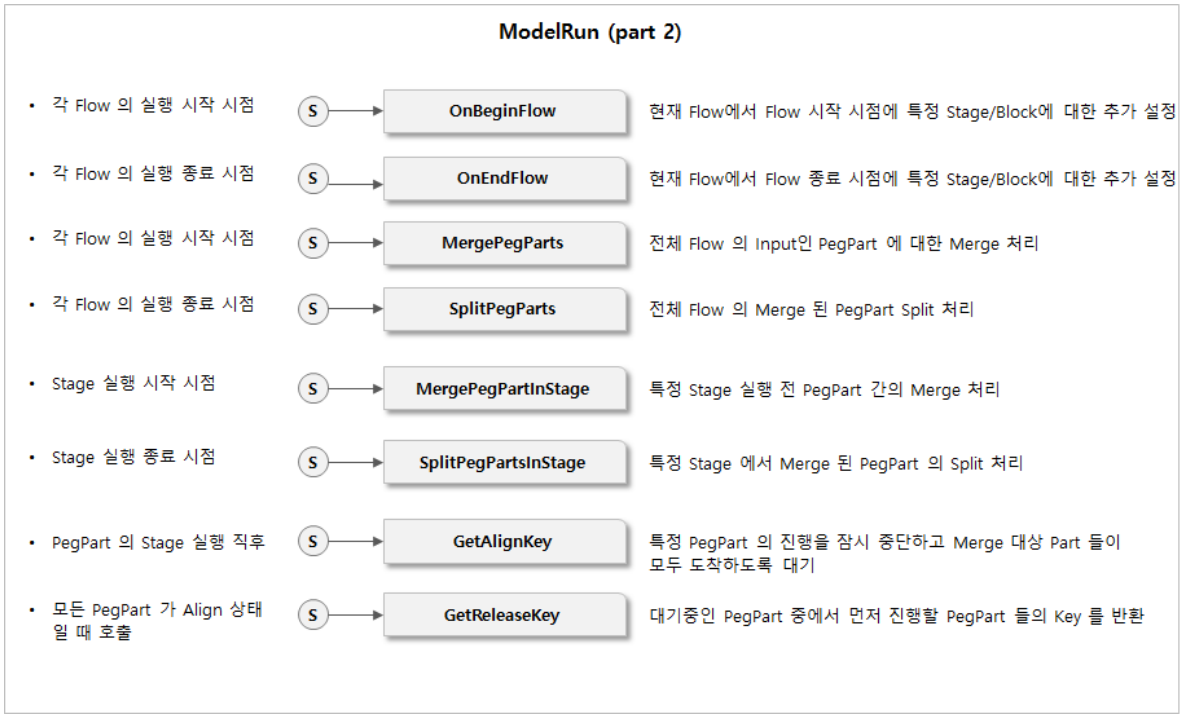
할 수 없습니다. 이러한 Logic 을 구현하기 위해 PegPart 의 우선순위를 평가하는 Logic 을 구현합니다.

> **Note**
> 상기 그림의 경우 3번 공정에서 PRODUCT1 과 PRODUCT2 로 분기하는 제품 구조를 나타냅니다. 상기 그림과 같은 제품의 경우 PRODUCT1, 2 에 대한 DEMAND 가 있을 경우 PEGGING 은 PRODUCT1의 6번 공정과 PRODUCT2의 8번 공정부터 한 공정씩 앞으로 전진하면서 진행합니다. 그런 경우 PRODUCT1 에 대한 PegPart 가 먼저 3번 공정에 도달하게 되는데, PegPart 의 CurrentStage 가 3번인 경우, PRODUCT1 에 대한 PegPart 는 PRODUCT2의 4번, 5번 공정에 대한 PEGGING 을 수행하기 전까지는 수행되어서는 안됩니다. 왜냐하면 PRODUCT2 가 4번 공정의 PEGGING 을 마치고 3번 공정에서 PRODUCT1로 MERGE 가 되어 함께 PEGGING 을 수행해야 하기 때문입니다.

구현 예시

```
public int COMPAREPEGPART(Mozart.SePlan.Pegging.PegPart x, PegPart y, ref bool handled,
int prevReturnValue)
{
    // 위의 그림의 경우처럼 모든 제품이 공정 3번에서 분기를 한다고 가정할 경우
    // 입력 값인 두 개의 PegPart x, y 의 Current Stage 에 따라 다음과 같은 Logic 을 구현할 수 있습니
다.
    // x 의 우선순위가 높은 경우 1, 동일한 경우 0, x 의 우선순위가 낮은 경우 -1 을 반환하도록 Logic 을 구
성합니다.
    if (x.CurrentStage.StageID == "3")
        return -1;
    if (y.CurrentStage.StageID == "3")
        return 1;
    return 0;
}
```

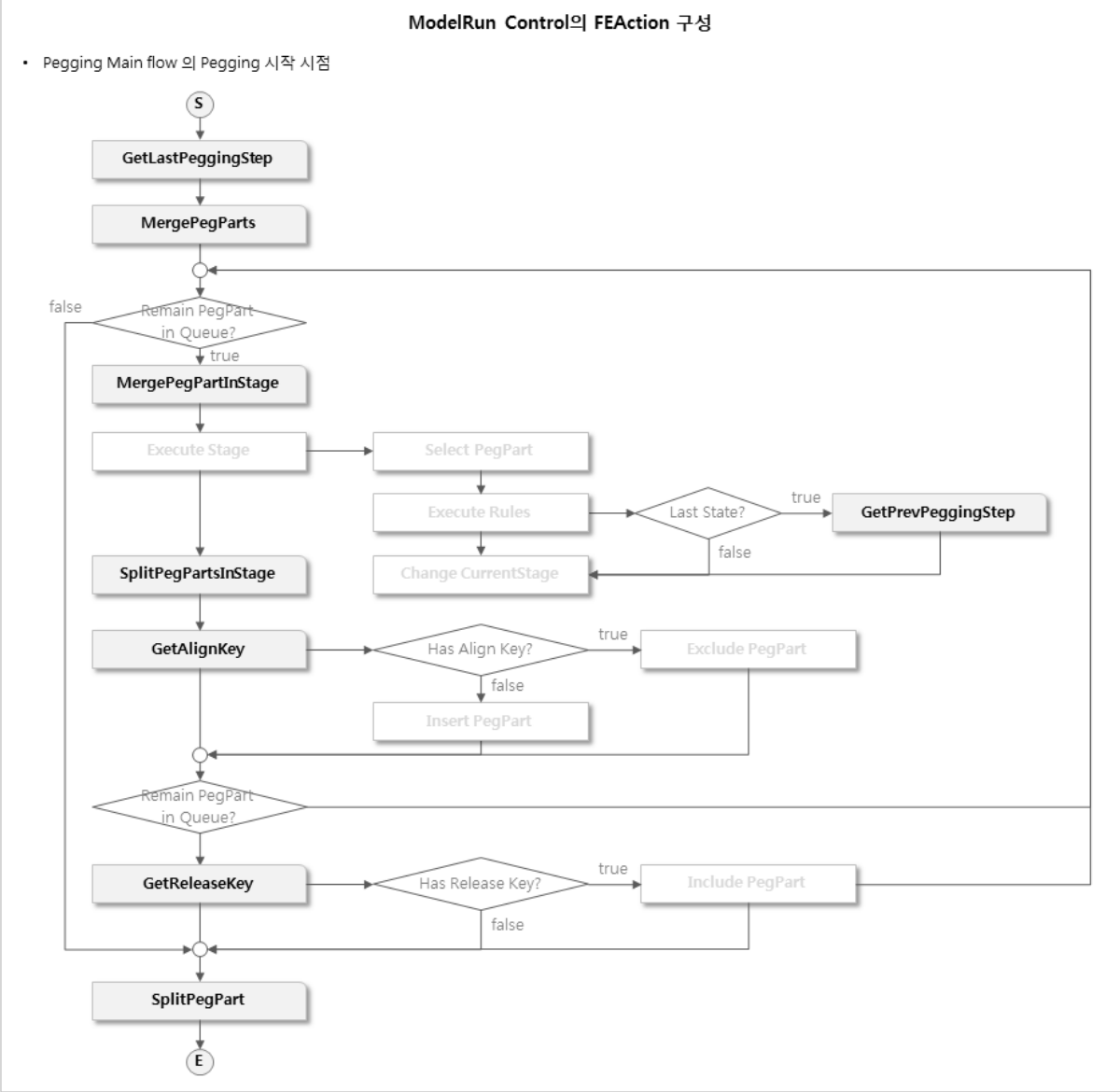
- **ComparePegTarget** : 특정 공정에서 Pegging 을 실행하는 시점에 각 동일 PegPart 내의 PegTarget 간의 Pegging 우선순위 Sorting 을 위한 비교 Logic 을 구현합니다.



- **OnBeginFlow** : Flow 실행시작 시점에 조건에 따라 특정 Stage/Block의 실행을 Enable/Disable 처리 등, Stage/Block에 대한 추가적인 설정을 할 수 있습니다.
- **OnEndFlow** : Flow 종료 시점에 조건에 따라 특정 Stage/Block의 실행을 Enable/Disable 처리 등, Stage/Block에 대한 추가적인 설정을 할 수 있습니다.
- **MergePegParts** : Flow 실행 시작시점에 개별 Demand (PegPart) 별 Stage Logic 처리를 하지 않고 특정 Logic (기준)에 따라 PegPart 의 묶음으로 처리하고자 하는 경우에 PegPart 를 MergedPegPart Type으로 Merge 하도록 Logic 을 구현합니다. 예를 들어 Pegger Model 의 Pre, Post flow 의 경우 Demand 별 Logic 을 처리할 필요가 없으므로 모든 PegPart 들을 하나의 PegPart 로 묶어서 처리할 수 있습니다.
- **SplitPegParts** : Flow 종료시점에 Merge 된 PegPart 가 존재하는 경우 이를 다시 PegPart 형태로 혹은 목적에 따라 또 다른 묶음 단위로 PegPart 를 나누는 Logic 을 구현합니다.
- **MergePegPartsInStage** : 특정 Stage 실행 전에 해당 Stage 에서 Merge되어 Pegging 될 필요가 있는 PegPart 를 묶어주는 Logic 을 구현합니다. 예를 들어 일부 재공의 경우 특정 Stage 에서 Product A 에 해당하는 제품의 재공이지만, A 에 대한 Demand 가 없는 경우 Product B, Product C 의 순으로 제품이 변경되어 Pegging 가능하다고 할 때 상기 Rule 의 구현을 위해 해당 Stage 에서 Product A, B, C 를 Merge 하고 Pegging 하는 Logic 에서 이를 처리하도록 구현할 수 있습니다.
- **SplitPegPartsInStage** : 특정 Stage 실행 후 다음 Stage 시작 전에 해당 Stage 에서 Merge 된 PegPart 를 유지하거나 특정 Logic 으로 나누는 Logic 을 구현합니다.

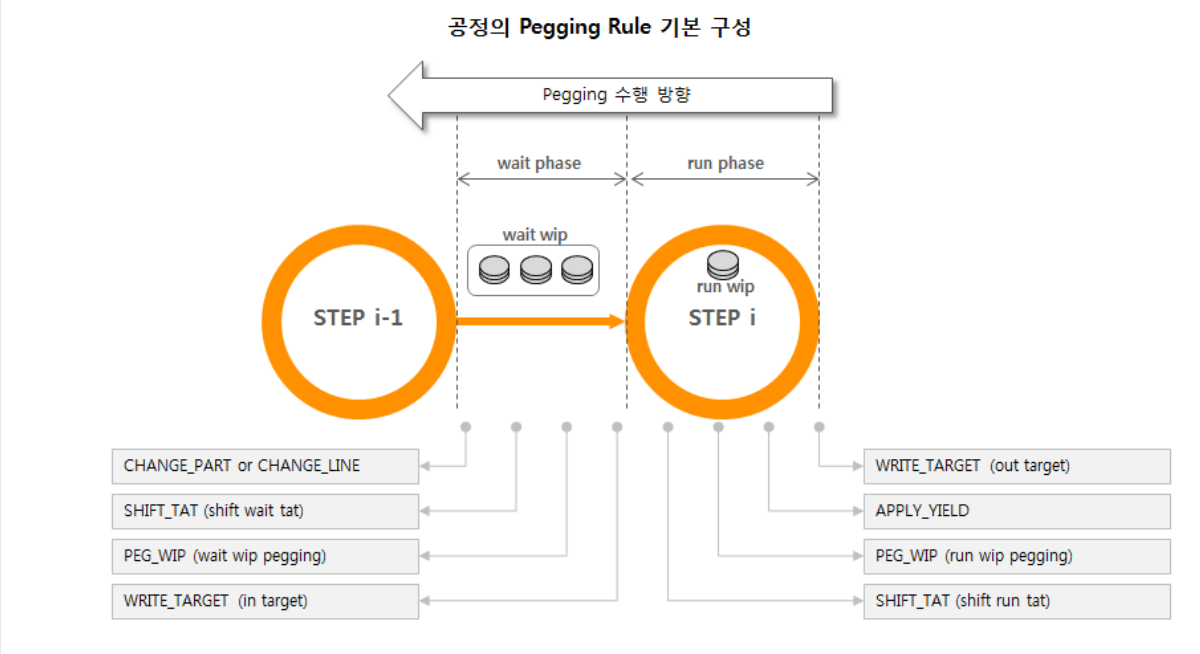
- **GetAlignKey** : PegPart 의 특정 Stage 가 종료한 직후에 호출됩니다. 만일 해당 Stage 가 끝나고 다음 Stage 에서 다른 PegPart 와 Merge 가 되는 경우에는 다른 PegPart 가 도착할 때까지 Pegging 을 진행하지 않고 대기하고 있어야 합니다. 이러한 경우에 Merge 할 제품코드 등의 Key 를 설정해주면 PegPart 는 대기 Queue 에 들어가고 실행이 중단 됩니다. 실행 대상이 되는 PegPart 가 더 이상 존재하지 않는 경우에 GetReleaseKey 를 통해 다시 진행할 PegPart 들의 Key 를 반환받아 진행하게 됩니다.
- **GetReleaseKey** : 정상적인 실행 대상 PegPart 가 더 이상 없고 Align (동일 제품 Merge) 를 위해 대기 중인 PegPart 가 있는 경우에 호출됩니다. 대기 중인 PegPart 중에 어떤 제품을 다시 진행할지 선택합니다. 선택하는 방식은 Align 시 지정된 Key 를 반환하는 방식입니다. Release 된 Key 가 동일한 PegParts 에 대해서는 MegePegPartInStage 함수를 구현하여 PegPart 를 Merge 처리합니다.

Model Control 의 각 FEAction 들은 아래와 같은 순서대로 Pegging 을 수행합니다.

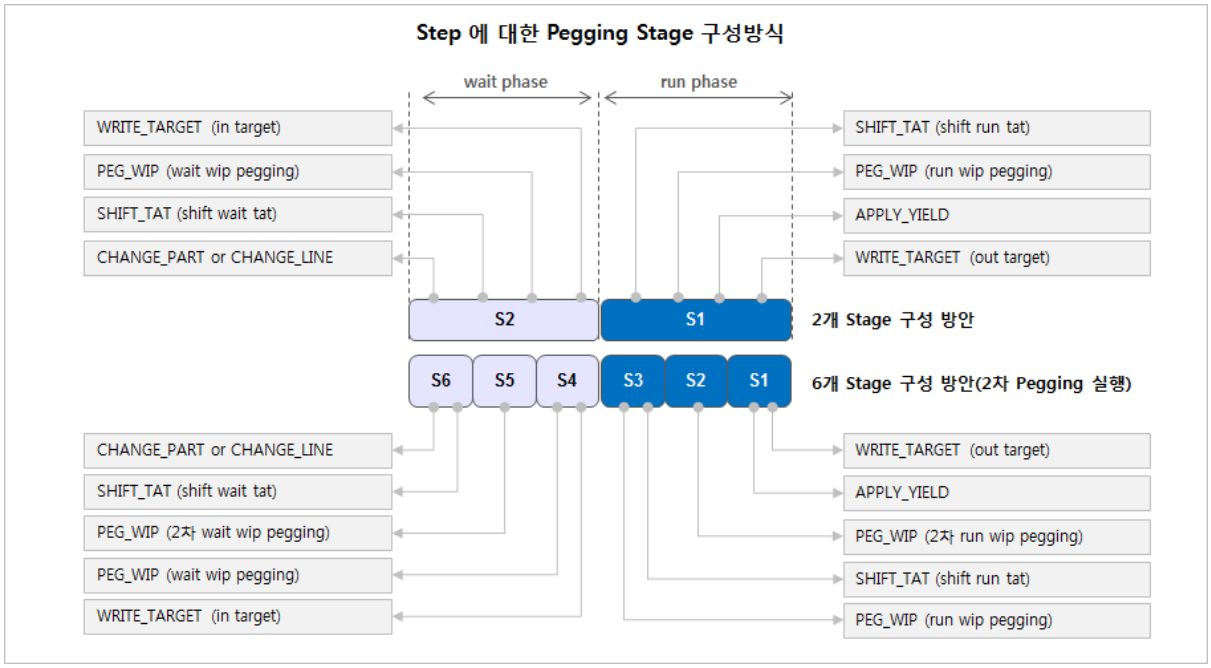


Rules

SeePlan Pegging Library 에서 제공하는 사전 정의된 Rule 로써 Stage 의 Rule 실행순서를 정의할 때 사용가능합니다. 사전 정의된 Rule 의 경우, Rule 의 실행에 필요한 Data 제공 및 Output 처리 Logic 등은 해당 Rule 에서 사용자가 정의할 수 있도록 구성되어 있습니다. Rule 별 Logic 의 구성과 사용자 정의 Logic 구현 Point (FEAction) 에 대한 설명을 제공합니다.



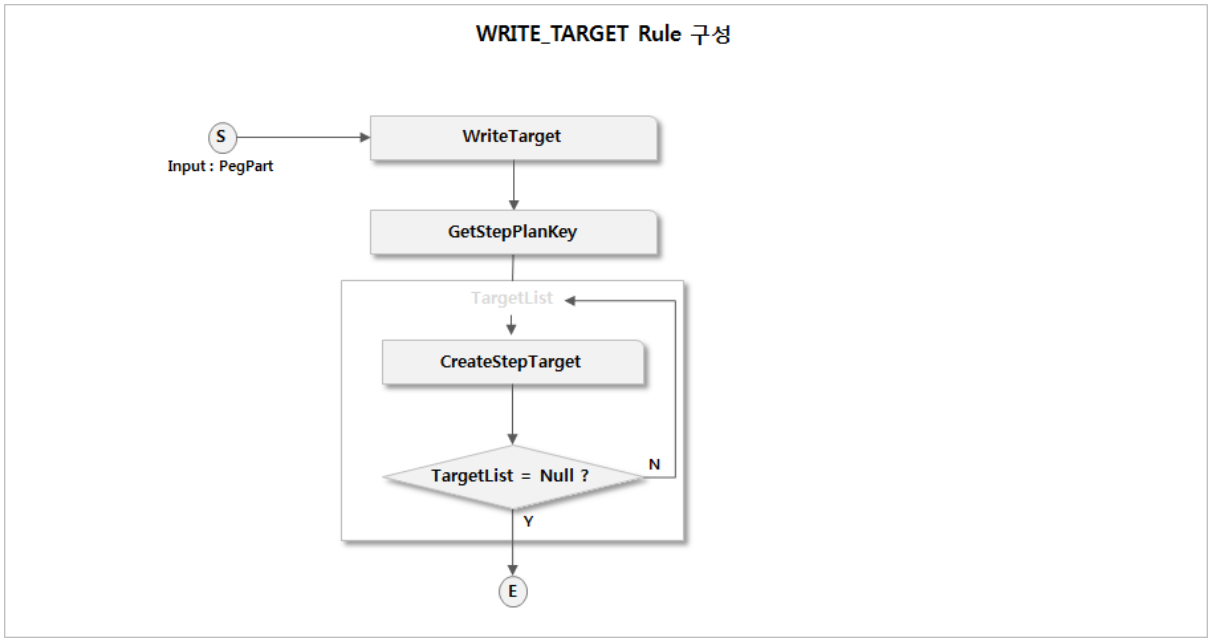
Step 은 상기와 같이 구성되어 있으나, Pegging 을 위해 Rule 을 배치하기 위한 Stage 는 Step 의 Phase 별로 복수 개로 구성이 가능합니다. 아래 그림은 사용자 조직의 Pegging 요구 사항에 따라 Step 별 Stage 를 다르게 구성하는 방식을 설명합니다.



위의 방식은 Run, Wait 별 하나의 Stage 에 Rule 을 구성하는 방식이며, 아래의 방식은 Run, Wait Pegging 을 여러 번 수행하기 위해 Stage 의 실행 Rule 을 더 작게 나누어 구성한 방식입니다. 이론적으로 Stage 는 사용자 조직의 Pegging Logic 에 맞게 n개로 구성할 수 있습니다.

WRITE_TARGET

특정 공정에서의 Demand 별 Target 정보를 기록하기 위한 Rule 입니다. 공정의 Target Datat 형식은 사용자 조직에 따라 다를 수 있음으로 Output 처리 부분에 대해 사용자가 정의할 수 있도록 FEAction 을 제공합니다.



- **WriteTarget** : Output 에 맞도록 Target 정보를 기록하는 코드를 작성합니다.

  구현 예시

```
public void WRITETARGET(Mozart.SePlan.Pegging.PegPart pegPart, bool isOut, ref bool
handled)
{
    SemiconPegPart pp = pegPart as SemiconPegPart;
    foreach (SemiconPegTarget ms in pp.PegTargetList)
    {
    WriteTarget(ms, pp.CurrentStage, pp.Product.ProductID, isOut);
    }
}
private void WriteTarget(SemiconPegTarget ms, PegStage stage, string prodID, bool is
Out)
{
    if (OutputMart.Instance.StepTarget == null)
    return;
    DateTime shiftDt = DateTime.MinValue;
    shiftDt = ShopCalendar.ShiftStartTimeOfDayT(ms.CalcDate);
    MfgStep step = stage.Tag as MfgStep;
      MfgProcess proc = step.Process as MfgProcess;
    StepTarget st = new StepTarget();
    st.TARGET_DATE = shiftDt;
    st.IN_TARGET_QTY = 0;
    st.OUT_TARGET_QTY = 0;
    OutputMart.Instance.StepTarget.Add(st);
    st.LINE_ID = proc.LineID;
    st.PRODUCT_ID = prodID;
    st.PROCESS_ID = step.Process.ProcessID;
    st.STEP_ID = step.StepID;
    if (isOut)
    {
        st.OUT_TARGET_QTY += ms.CalcQty;
```

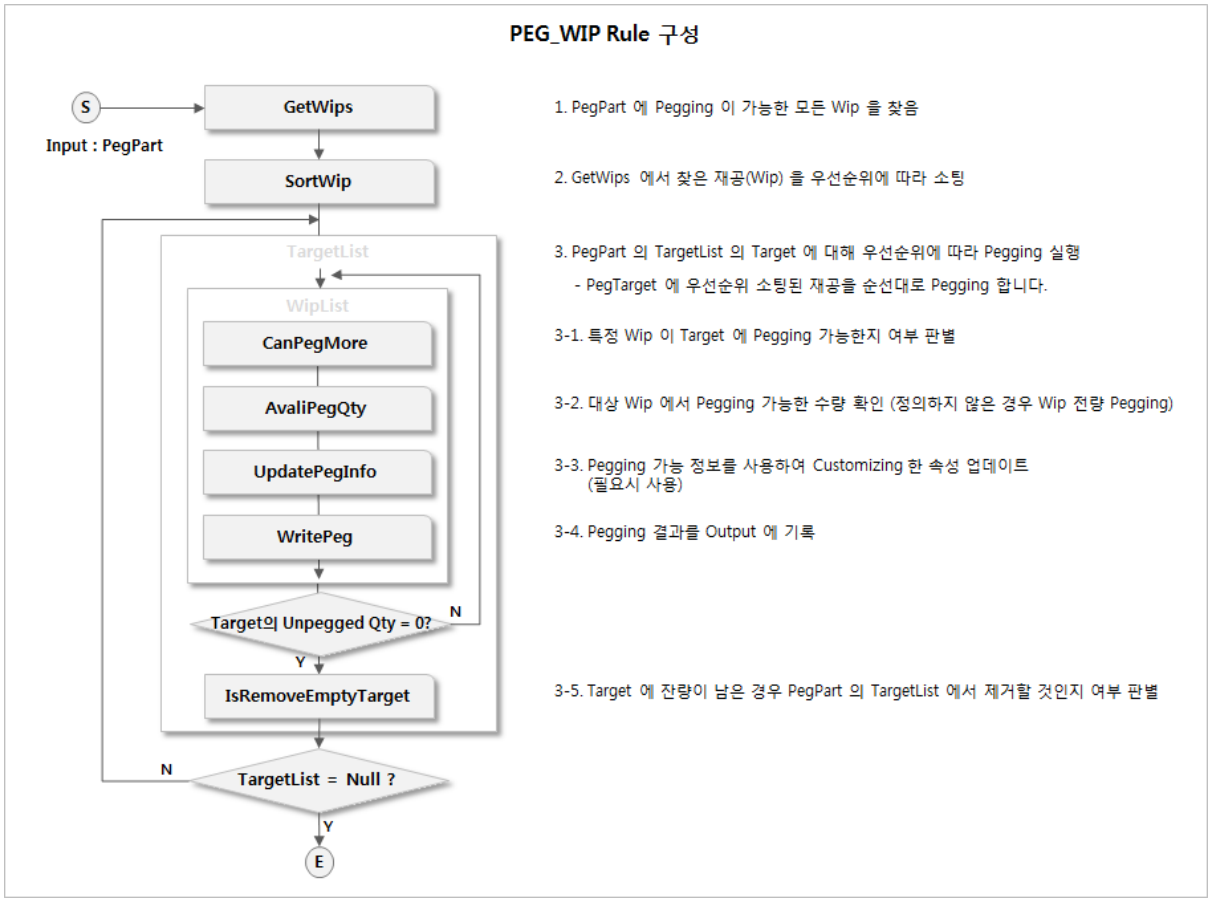
```
    }
    else
    {
        st.IN_TARGET_QTY += ms.CalcQty;
    }
    st.MO_DEMAND_ID = ms.Mo.DemandID;
    st.MO_DUE_DATE = ms.Mo.DueDate;
    st.MO_PRIORITY = 1;
    st.MO_PRODUCT_ID = ms.Mo.ProductID;
}
```

- **GetStepPlanKey** : Backward Pegging 실행 후 Simulation 을 수행하면서 Forward Pegging 기능을 사용하는 경우에 사용합니다. PegTarget 으로부터 StepPlan 에 StepTarget 을 기록할 때 동일 Step 내에서 StepPlan 을 구분하기 위한 Key 를 반환합니다. null 을 반환하는 경우 StepPlan 에 StepTarget 을 등록하지 않습니다. 일반적으로 StepPlanKey 는 제품코드가 되는 경우가 많습니다.
- **CreateStepTarget** : PegTarget 의 정보와 Key 를 기준으로 Forward Pegging 에서 사용할 StepTarget 을 생성합니다. 아래는 상기 GetStepPlanKey, CreatStepTarget 을 통해 Forward Pegging 을 준비하는 예제입니다.

```
//Forward Pegging 을 위한 StepPlan 생성예제
// GetStepPlanKey 함수 구현
public object GETSTEPPLANKEY(PegPart pegPart, ref bool handled, object prevReturnVal
ue)
{
    // 제품객체를 Key 로 사용
    return (pegPart as SitePegPart).Product;
}
// CreateStepTarget 함수 구현
public StepTarget CREATESTEPTARGET(PegTarget pegTarget, object stepPlanKey, Step ste
p, bool isRun, ref bool handled, StepTarget prevReturnValue)
{
    // Library 의 기본 클래스 생성
    var pt = pegTarget as SitePegTarget;
    var st = new StepTarget(stepPlanKey, step, pt.Qty, pt.DueDate, isRun);
    return st;
}
```

PEG_WIP

특정 공정에서의 주어진 Demand 에 대해 Pegging 대상 Wip 정보를 찾아 Pegging 처리를 수행하는 Rule 입니다. Rule 의 실행을 위해 아래의 Logic (FEAction) 을 사용자가 정의해야 합니다.



- **GetWips** : 주어진 Demand 에 Pegging 이 가능한 재공을 반환하도록 Logic 을 작성합니다. Pegging 조건을 확인하고 Stage 의 상태에 따라 대상 재공 List 를 반환합니다. Pegging 제약을 반영할 수 있습니다.

  구현 예시

```
public IList<Mozart.SeePlan.Pegging.IMaterial> MAP_WIP(Mozart.SeePlan.Pegging.PegPar
t pegPart, bool isRun, ref bool handled, IList<IMaterial> prevReturnValue)
{
    SemiconPegPart pp = PegHelper.GetPlanPart(pegPart);
    MfgStep step = PegHelper.GetCurrentStep(pegPart);
    HashSet<PlanWip> wips;
    List<IMaterial> result = new List<IMaterial>();
    if (InputMart.Instance.PlanWips.TryGetValue(step, out wips) && wips.Count > 0)
    {
        foreach (PlanWip wip in wips)
        {
            if (isRun != (wip.State == "RUN"))
            continue;
            if (wip.Wip.Product.ProductID == pp.Product.ProductID)
            {
                wip.MapCount++;
                result.Add(wip);
            }
        }
    }
```

```
    return result;
}
```

- **SortWip** : GetWips 를 통해 선택된 재공에 대해 Pegging 우선순위가 높은 순서로 재공을 Sorting 합니다. 구현하지 않으면 재공의 경우 Random 하게 Pegging 이 됩니다.
- **CanPegMore** : 개별 재공별로 Demand 에 대해 Pegging 이 가능한지 여부를 판단하는 Logic 을 작성합니다. 재공의 속성 및 Demand 의 상태에 따라 Pegging 을 제약할 수 있는 요소를 Logic 에 반영할 수 있습니다.
- **AvailPegQty** : 특정 Lot 을 PegTarget 에 Pegging 할 때, 경우에 따라 Lot의 Unit 전량을 Pegging 하지 않고 일부만 Pegging 을 할 수 있습니다. 이 경우 Lot 의 Pegging 가능 수량을 반환하는 Logic 을 구현합니다.
- **UpdatePegInfo** : Pegging 시점에 PegTarget, PegPart 의 속성을 변경하고자 하는 경우 사용합니다. PegTarget 에 기본적으로 잔여 수량 정보가 업데이트됩니다. 다만, Pegging 된 수량이나 Lot 정보를 사용해서 사용자가 정의한 Data 의 속성을 가공하는 경우에는 이를 본 함수에서 구현합니다. 예를 들어 PegTarget 에 사용자가 수율이 반영되지 않은 잔여 수량 정보를 가공하고자 하는 경우에 이러한 속성을 PegTarget 에 정의하고 본 함수에서 이 속성을 업데이트 할 수 있습니다.
- **WritePeg** : 재공별로 Pegging 이 된 경우, Pegging 결과를 사용자 조직에서 정의한 Pegging Output 에 따라 기록합니다.

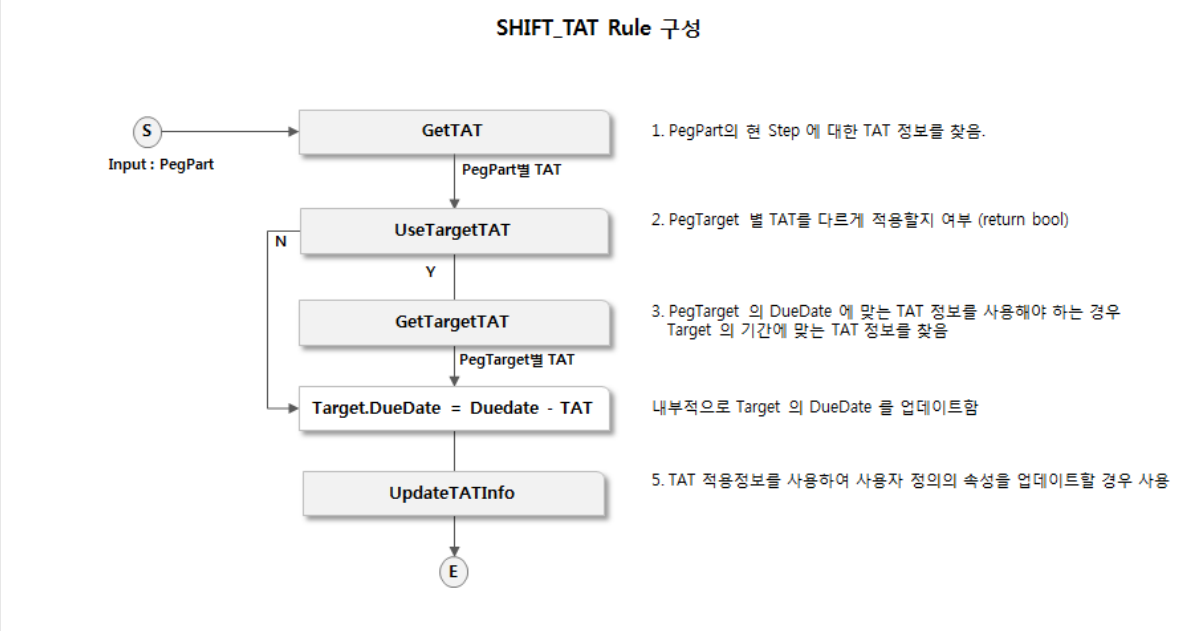
  구현 예시

```
public void WRITEPEG(PegTarget target, IMaterial m, float qty, ref bool handled)
{
    if (OutputMart.Instance.PegHistory == null)
        return;
    SemiconPegTarget ms = target as SemiconPegTarget;
    PlanWip wip = material as PlanWip;
    DateTime shiftDt = ShopCalendar.StartTimeOfDayT(ms.CalcDate);
    MfgStep step = PegHelper.GetCurrentStep(target.PegPart);
    MfgProcess proc = step.Process as MfgProcess;
    PegHistory history = new PegHistory()
    {
        LINE_ID = (wip.Wip as WipInfo).LineID,
        LOT_ID = wip.Wip.LotID,
        PRODUCT_ID = wip.Wip.Product.ProductID,
        STEP_ID = (wip.Wip as WipInfo).Step.StepID,
        LOT_QTY = wip.Wip.UnitQty,
        PEG_QTY = qty,
        MO_DEMAND_ID = ms.Mo.DemandID,
        MO_DUEDATE = ms.Mo.DueDate,
        MO_PRODUCT_ID = ms.ProductID,
        LPST = ms.CalcDate
    };
    OutputMart.Instance.PegHistory.Add(history);
}
```

- **IsRemoveEmptyTarget** : Pegging 된 이후 남은 Target 잔량이 없는 경우, PegPart 리스트에서 삭제 될 수 있게 설정합니다. 초기값은 false 로 PegTarget이 삭제되지 않습니다.

SHIFT_TAT

공정의 TAT만큼 Target 의 시간을 이동하는 Logic 을 실행합니다. 공정에서의 Run phase, Wait phase 를 고려하여 해당 TAT 를 반환하는 Logic 을 구현합니다. 사이트에 따라 TAT 적용하는 Rule 를 다르게 구성할 경우 Run 혹은 Wait TAT 만 고려한다거나 TAT 를 고려하는 순서와 위치를 변경해서 사용할 수 있습니다.



- **GetTat** : 사용시점의 PegPart 의 pegging 대상 공정(CurrentStage)에 대한 TAT 정보를 반환하는 코드를 작성합니다.

  구현 예시

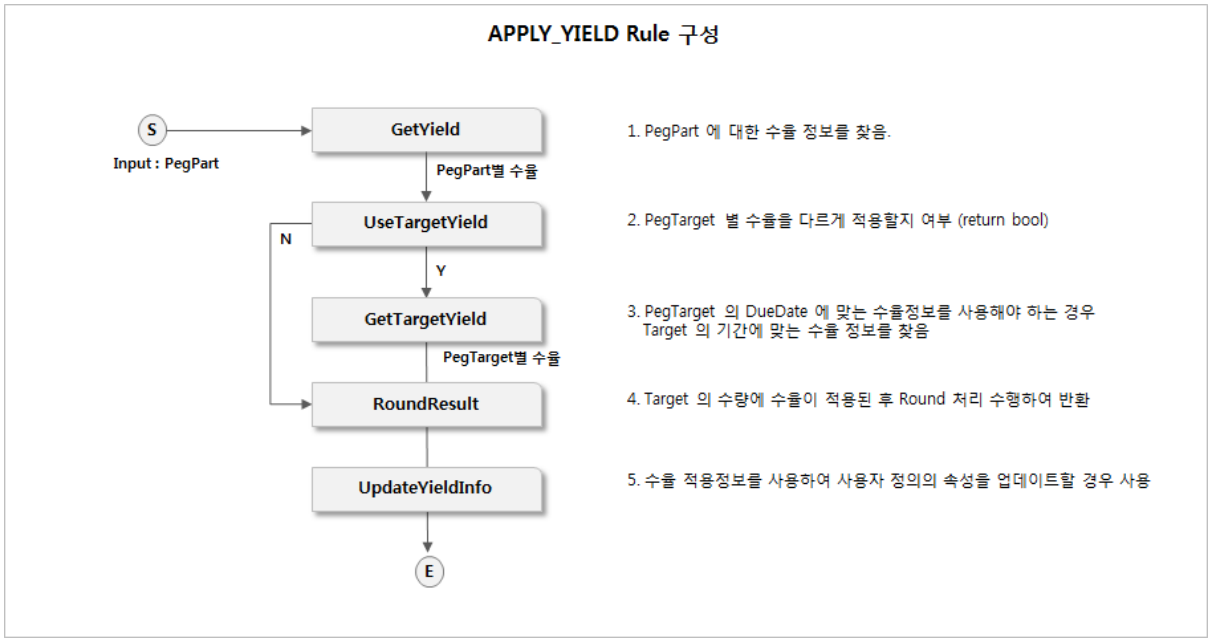
```
public TimeSpan GETTAT(Mozart.SePlan.Pegging.PegPart pegPart, bool isRun, ref bool
 handled, TimeSpan prevReturnValue)
{
    SemiconPegPart pp = pegPart as SemiconPegPart;
    MfgStep step = pp.CurrentStage.Tag as MfgStep;
    StepTat tat = InputMart.Instance.StepTat.Rows.Find(new object[] { pp.Product.Lin
eID, pp.Product.ProductID, step.StepID });
    double t = double.MinValue;
    if (tat != null)
    {
        if (isRun)
```

```
            t = tat.RUN_TAT;
        else
            t = tat.WAIT_TAT;
    }
    else
        t = 3600d; //기본 TAT 1Hr
    return TimeSpan.FromSeconds(t);
}
```

- **UseTargetTAT** : 동일 Product(PegPart) 내의 Target 별로 적용되는 TAT 의 차등을 두어 적용하고자 하는 경우 True 반환, 그렇지 않은 경우 False. 기본값은 False 임. True 인 경우 성능에 영향을 줄 수 있습니다.
- **GetTargetTAT** : 동일 Product 의 동일 Step 에 대한 TAT 라도 시간에 따라 적용되는 TAT 가 다른 경우가 있을 수 있습니다. 이 때에는 PegTarget 별로 TAT 를 설정할 수 있으며, 본함수에서 해당 내용을 처리합니다. 본 함수를 구현하고 값이 있는 경우 GetTat 에서 반환한 값보다 우선 적용됩다.
- **UpdateTatInfo** : TAT 를 업데이트 한 직후 호출되는 Action 입니다. TAT 를 반환한 경우 Library 에서 Target의 DueDate 를 자동으로 업데이트합니다. DueDate 이외의 사용자 정의 속성에 대한 변경이나 사용자 Logic 을 구현 할 수 있습니다.

APPLY_YIELD

공정에 대한 수율을 반영하는 Logic 을 처리합니다. 수율이 90%인 경우 Out target 이 100인 공정의 경우 In Target = 100/0.9 = 111 로 수율을 고려한 Target을 생성해야 하며, 본 Rule 에서 해당 Logic 을 처리합니다. 일반적으로 Demand 에 평균적인 수율을 고려하여 수정한 경우에는 공정별 수율을 반영하지 않으며, 공정별 수율적용이 필요한 경우에 Rule 을 적용합니다.



Get Started

- **GetYield** : 공정에 적용할 수율을 반환합니다. 공정별 수율을 사용하는 경우에 구현이 필요하며, 기본값은 1로 구현하지 않을시에는 수율이 반영되지 않는 것으로 모델링하는 결과를 만듭니다.

구현 예시

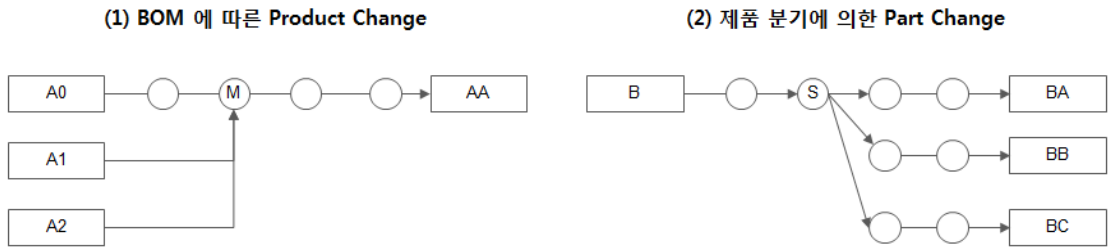
```
public float GETYIELD(Mozart.SeePlan.Pegging.PegPart pegPart, ref bool handled, float pr
evReturnValue)
{
    SemiconPegPart pp = PegHelper.GetPlanPart(pegPart);
    MfgStep current = PegHelper.GetCurrentStep(pegPart);
    Yield y = InputMart.Instance.Yield.Rows.Find(new object[] { pp.Product.LineID, pp.Pr
oduct.ProductID, current.StepID });
    if (y != null)
        return (float)y.STEP_YIELD;
    return 1.0f;
}
```

- **UseTargetTAT** : 동일 Product(PegPart) 내의 Target 별로 적용되는 수율의 차등을 두어 적용하고자 하는 경우 True 반환, 그렇지 않은 경우 False. 기본값은 False 임. True 인 경우 성능에 영향을 줄 수 있습니다.
- **GetTargetYield** : 기간별로 적용하는 수율에 변동이 있는 경우는 Target 별로 다른 수율을 적용할 수 있습니다. 이 경우에 본 함수를 구현합니다. 예를 들어 Duedate 가 4월인 Target의 경우 수율을 95% 로 적용하고 5월에 98%를 적용한다고 할 때, Target 의 Duedate 를 확인하고 다른 수율을 반환하도록 처리할 수 있습니다.

- **RoundResult** : 수율이 반영된 최종 결과 값에 대해 올림, 반올림 등의 처리를 하여 수율이 반영된 최종 결과값을 반환합니다.
- **UpdateYieldInfo** : 수율 반영된 정보를 기반으로 사용자가 필요한 정보를 업데이트하기 위한 Action 입니다.

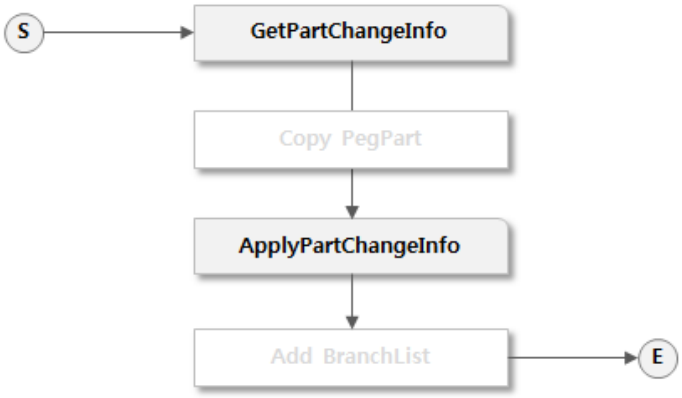
CHANGE_PART

Product Route 에 의해 제품의 변경이 가능한 경우 제품변경이 발생하는 공정에서 다음공정의 Pegging 을 하기 전에 변경대상(To) Product 정보를 찾아 반환하는 Logic 을 구성합니다. Product 가 변경되는 경우는 크게 ***1) 제품의 BOM 구조에 의해 변경되는 경우***와 ***2) 공용제품에서 품질이나 고객에 따라 제품이 분기되어 변하는 경우***로 나뉠 수 있습니다. 본 Rule 은 제품변경에 대한 모델링이 필요한 경우 사용됩니다.



(1) BOM 에 Product Change 의 경우에는 AA 제품에 대한 Demand 가 있는 경우 공정 M 에서 A0, A1, A2 제품에 대한 Demand 를 추가로 생성해주어야 하며, (2) 제품분기의 경우 각 BA, BB, BC 제품에 대한 Demand 가 공정 S 에서 B 제품의 Demand 로 변경하여 해당 제품의 재공에 Pegging 을 수행해야 합니다. 즉 이러한 Pegging 특성을 고려하여 CHANGE_PART RULE 은 아래와 같은 두 가지 FEAction을 통해 사용자가 상황에 따른 제품 변경을 모델링 할 수 있습니다.

CHANGE_PART Rule 구성



- **GetPartChangeInfo** : PegPart 의 현재 공정 이후에 변경되어야 하는 Product 정보를 반환합니다.

구현 예시

```
public List<object> GETPARTCHANGEINFOS(PegPart pegPart, ref bool handled, List<object> p
revReturnValue)
{
    SemiconPegPart pp = (SemiconPegPart)pegPart;
    string currentLineID = pp.Product.LineID;
    string currentProdID = pp.Product.ProductID;
    string currentProcID = pp.Product.Process.ProcessID;
    string currentStepID = (pp.CurrentStage.Tag as MfgStep).StepID;
    var infos = InputMart.Instance.ProductRoute.Rows.Where(p => (p.LINE_ID == currentLin
eID &&
        p.TO_PROD_ID == currentProdID &&
        p.TO_PROC_ID == currentProcID &&
        p.TO_STEP_ID == currentStepID)
        );
    var list = infos.ToList<object>();
    return infos.ToList<object>();
}
```

- **ApplyPartChangeInfo** : 제품 변경정보에 따라 생성된 PegPart 에 변경되는 Product 및 공정(Step) 정보를 설정합니다.

  구현 예시

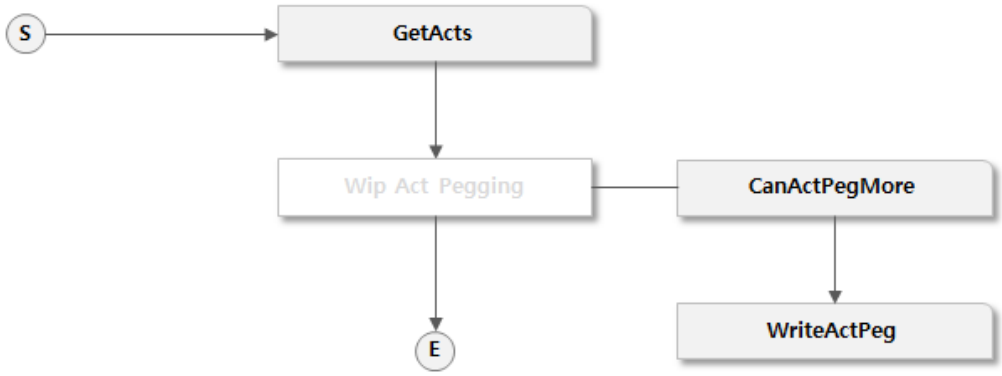
```
public PegPart APPLYPARTCHANGEINFO(PegPart pegPart, object partChangeInfo, ref bool
 handled, PegPart prevReturnValue)
{
    ProductRoute route = (ProductRoute)partChangeInfo;
```

```
    MfgProduct changeProd = InputMart.Instance.ProductList[route.FR_PROD_ID];
    MfgProcess changeProc = InputMart.Instance.ProcessList[route.FR_PROC_ID];
    MfgStep changeStep = (MfgStep)changeProc.FindStep(route.FR_STEP_ID);
    PegStage changeStage = pegPart.CurrentStage.Model.GetStage(changeStep);
    SemiconPegPart pp = (SemiconPegPart)pegPart;
    pp.Product = changeProd;
    pp.CurrentStage = changeStage;
    return pp;
}
```

APPLY_ACT

초기 Demand 에 대한 실적을 차감하기 위해 사용되는 Rule 입니다. Pegging 을 수행하는 시점에 Demand 기간 내에 이미 생산 완료된 Demand 를 차감하여 실제 생산이 필요한 Demand 에 대해서 Pegging 을 할 수 있도록 합니다.

APPLY_ACT Rule 구성



- **GetActs** : Demand 에서 차감할 실적정보를 반환합니다. 반환할 실적정보는 IMaterial Interface 를 구현합니다.
- **CanActPegMore** : 대상 실적정보를 현재 Demand (PegPart) 에 Pegging 가능한지 여부를 판별하는 Logic 을 구현합니다.
- **WriteActPeg** : 대상 실적정보가 현재 Demand (PegPart) 에 Pegging 된 경우, 이에 대한 Pegging 결과를 사용자 조직에서 정의한 Output Format 으로 기록합니다.

MANIPULATE_DEMAND

Demand 입력정보를 Pegging 을 수행하기 위해 일자별로 동일 수준으로 맞추어 나누어주거나 납기를 기준으로 조정이 필요한 경우 본 Rule 에서 상기 Logic 을 처리합니다. 일반적으로

Pegger Model 의 Preflow 를 구성하는 Stage 에 포함되는 Rule 로 구성합니다.

- **ManipulateDemand** : 입력된 Demand 를 Pegging 의 Input Demand 로 변환하는 Logic 을 구현합니다.

PREPARE_TARGET

Demand 에 대한 정보를 기반으로 PegTarget 정보를 생성합니다. 일반적으로 Out 기준 Demand 에 Mapping 되는 제품 (Product) 별로 PegPart 를 생성합니다.

- **PrepareTarget** : Pegging 수행을 위한 Demand 정보를 기반으로 PegTarget 을 생성하여 반환합니다.

구현 예시

```
public PegPart PREPARETARGET(PegPart pegPart, ref bool handled, PegPart prevReturnValue)
{
    MergedPegPart mg = pegPart as MergedPegPart;
    foreach (SemiconMoMaster mm in InputMart.Instance.Demands.Values)
    {
        SemiconPegPart pp = new SemiconPegPart(mm, mm.Product);
        foreach (SemiconMoPlan mp in pp.MoMaster.MoPlanList)
        {
            SemiconPegTarget ms = new SemiconPegTarget(pp, mp);
            pp.AddPegTarget(ms);
        }
        mg.Merge(pp);
    }
    return (mg as PegPart);
}
```

PREPARE_WIP

초기 재공 (Wip) Data 로 Pegging 을 위한 PlanWip 정보를 생성합니다. 이때 Wip 정보에 대한 Validation 처리를 할 수 있으며, 초기 Wip 이 위치한 공정(Step) 에 대한 조정을 수행할 수 있습니다.

- **PrepareWip** : Pegging 을 위한 Wip 정보를 생성하고 초기화를 수행합니다.

구현 예시

```
public PegPart PREPAREWIP(PegPart pegPart, ref bool handled, PegPart prevReturnValue)
{
  HashSet<PlanWip> wipSets = null;
  foreach (WipInfo info in InputMart.Instance.WipList.Values)
  {
      PlanWip wip = new PlanWip(info);
      if ((wip.Wip as WipInfo).Process == null)
      {
          // there are no process info
          PegHelper.WriteUnpegHistory(wip, wip.Wip.UnitQty, "STD", "No Process Info");
          continue;
      }
      if ((wip.Wip as WipInfo).Step == null)
      {
          // there are no step info
          PegHelper.WriteUnpegHistory(wip, wip.Wip.UnitQty, "STD", "No Step Info");
          continue;
      }
      if (InputMart.Instance.PlanWips.TryGetValue(info.Step, out wipSets) == false)
      {
          wipSets = new HashedSet<PlanWip>();
          InputMart.Instance.PlanWips.Add(info.Step, wipSets);
      }
      wipSets.Add(wip);
  }
  return pegPart;
}
```

BUILD_INPLAN

Pegging 이 완료된 후 잔여 Demand 에 대해 투입계획을 수립합니다. 일반적으로 본 Rule 은 Pegger Model 의 Post flow 를 구성하는 Rule 로, 입력 값인 PegPart 가 MergedPegPart 입니다. 산출된 투입계획은 최종 Output 으로 사용되거나 Forward Simulation 수행 시의 투입 Lot 생성을 위한 기본 Input 으로 사용될 수 있습니다.

- **BuildInPlan** : PegPart 의 잔여 Target 을 투입 배치를 구성하는 기준으로 다시 Grouping 하고 사용자 조직의 투입 Rule 을 고려하여 투입계획을 만드는 Logic 을 개발합니다.

WRITE_UNPEG

Pegging 이 완료된 후 Pegging이 되지 않고 남아있는 재공에 대해 사유를 사용자 조직에서 필요한 수준으로 Output 에 기록합니다. 일반적으로 Pegging 일 완료된 후 Pegging 되지 않고 남아있는 재공은 재공의 제품에 대한 Demand 가 없거나, Demand 에 비해 재공이 너무 많이

투입된 경우로 볼 수 있습니다. 두 가지 경우에 대해 사용자 조직의 요구사항에 따라 좀 더 세분화된 사유를 분석하는 Logic 을 개발할 수 있습니다.

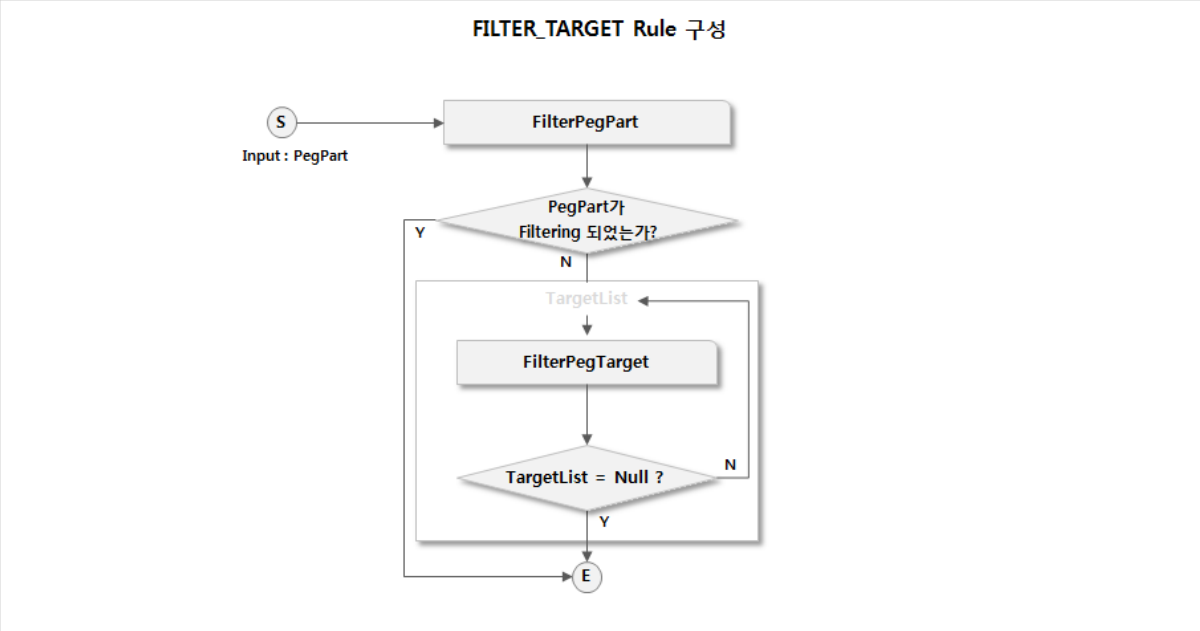
- **WriteUnpeg** : 잔여재공의 상태를 통해 Unpegging 사유를 Output 에 기록합니다.

구현 예시

```
public void WRITEUNPEG(Mozart.SeePlan.Pegging.PegPart pegPart, ref bool handled)
{
    foreach (HashSet<PlanWip> wipList in InputMart.Instance.PlanWips.Values)
    {
        foreach (PlanWip wip in wipList)
        {
            if (wip.Qty == 0)
            continue;
            if (wip.MapCount == 0)
            {
                PegHelper.WriteUnpegHistory(wip, wip.Qty, "PEG", "NO TARGET");
            }
            else if (wip.Qty > 0)
            {
                PegHelper.WriteUnpegHistory(wip, wip.Qty, "PEG", "EXCESS");
            }
        }
    }
}
```

FILTER_TARGET

PegPart 혹은 PegPart 가 포함하는 PegTarget 별로 Pegging 을 수행하지 않도록 설정하기 위해 사용합니다. 해당 Rule 의 실행 시 먼저 PegPart 자체를 Filtering 할지 판별하며, PegPart 가 Filtering 되지 않은 경우 PegTarget 별로 Filtering 여부를 판별하도록 기본 Logic 이 구현되어 있습니다. Rule 적용 시 아래의 Action이 노출되며 이를 통해 Filtering Logic 을 처리합니다.



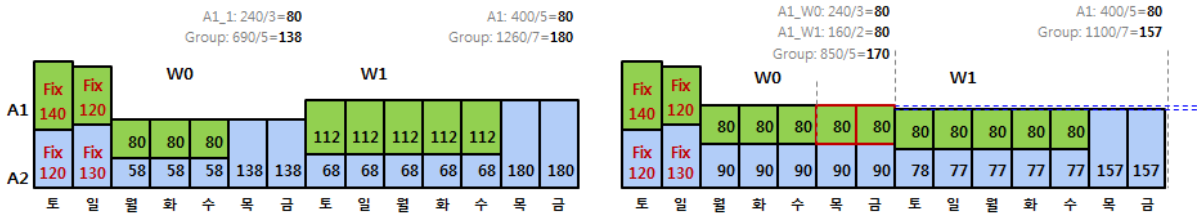
- **FilterPegPart** : 판별 대상 PegPart 를 현재 Step에서 Pegging 할지 여부를 판별합니다.
- **FilterPegTarget** : 판별 대상 PegTarget 을 현재 Step 에서 Pegging 할지 여부를 판별합니다. 예를 들어 특정 Step에서 특정 일자의 제품 Demand 에 대한 Pegging 이 금지되는 조건들이 있다면 이 함수에서 해당 Target 을 Filtering 하도록 구현할 수 있습니다.

SMOOTH_DEMAND

Pegging 을 위해 Demand 정보를 가공하는 Rule 입니다. Demand 정보는 주로 제품별 주/월 단위 요구수량의 형태로 구성됩니다. Pegging 을 하기 위해서 일반적으로 이러한 Demand 정보를 일별/제품별 생산요구수량 정보로 변환합니다. SMOOTH_DEMAND Rule 은 Demand 의 납기와 생산 Capacity 를 고려하여 원 Demand 를 일별 생산목표량으로 전환하는 작업을 위해 Library 에서 제공되는 Rule 입니다. 해당 Rule 의 실행을 위해서는 구현 시 Library 에서 제공하는 Smoother 클래스를 사용합니다.

기본적으로 Smoother 는 제품별 Demand 를 합하여 생산기간동안 매일 동일한 양을 생산할 수 있다는 전제하에 일별 생산 Capacity 를 정한 후 해당 Capacity Bucket 을 제품별 Demand 로 채워가면서 일별 생산목표를 만듭니다. 아래는 Demand Smoothing 에 대한 예시적인 개념을 표시합니다. 예시에서 Cut-Off 는 주차별 계획량을 주의 마지막 일로부터 얼만큼 당겨 진행해야하는지를 설정하는 속성입니다. Cut-Off = 2일 이라는 것은 한주가 월요일~일요일로 구분된다는 가정하에 금요일까지 목표생산량을 생산해야 한다는 의미입니다.

상기 1번에서 A1의 W0 목표량은 500이며 Cutoff 가 2일임으로 5일동안 생산해야 합니다. 즉 500/5 = 100, 일별로 100Unit 씩 생산해야 하는 목표가 나옵니다. A2의 경우 W0의 목표량이 700이며 Cut-off 가 0 이므로 7일동안 생산해야 합니다. 즉, 700/7 = 100, 일별로 동일하게 100 Unit 씩 생산해야 합니다. 두 제품모두 D0, D1을 확정계획만큼 생산해야 한다고 하면 확정계획을 제외한 양으로 일별 생산계획을 수립해야 하며 그 결과 확정계획 Mapping 결과처럼 결과가 나옵니다. 이를 제품 Group 단위로 합하여 일별 Capacity를 설정하면 아래 그림과 같은 결과가 생성됩니다.



아래는 SMOOTH_DEMAND Rule 의 사용 시 사용자가 구현해주어야 하는 항목을 표시합니다.

- **GetSmoother** : SMOOTH_DEMAND rule 에서 사용할 Smoother 클래스 객체를 생성하고 초기화하여 반환합니다. Smoother 클래스의 사용법은 Library Reference 의 해당 클래스를 참조하시기 바랍니다. Smoother 생성 시 주요 Input 은 아래와 같습니다.

> **Note**
> **MoPlanList** : Smoothing 전의 Demand 로 MoPlan List 입니다.
> **MoPlanComparerForSmoothing** : Demand Smoothing 을 하면서 설정된 Capacity Bucket 을 채우는 순서를 결정하기 위한 Comparer 입니다.
> **MoPlanComparerForPrioritizing** : 개별 Demand 에 대한 우선순위를 부여하기 위한 Comparer 입니다.
> **fixDays** : 기존 계획 중 확정계획을 반영해야하는 일 수입니다. 예를 들어 2로 설정하면, D0, D1 의 MoPlan 은 계획을 고정시킨 상태에서 Bucket 을 채우는 방식으로 진행합니다.
> **allowEarllyProcess** : Demand 에 포함된 계획을 계획보다 선행해서 생산이 가능한지 여부를 설정합니다. 기본값은 false 입니다. true 로 설정하면 선행해서 Demand 를 분배할 수 있습니다.
> **batchSize** : Bucket 에 Demand (MoPlan) 채워서 Capacity 를 차감할 때의 기본 단위입니다. 예를 들어 batchSize 가 200인 경우에는 200unit 단위로 Bucket 을 쌓아가는 방식이 됩니다.

- **GetInnerBucketKey** : 동일한 제품군 내에서 동일한 납기를 가진 Demand 에 대한 Group 을 InnerBucket 이라고 합니다. 본 Action에서 특정 Demand 가 어떤 InnerBucket 을 사용할지 판별하고 InnerBucket 에 대한 Key 를 반환합니다.
- **GetOuterBucketKey** : 복수 개의 제품군을 Grouping 하여 Capacity 할당을 하기 위한 Group 에 대한 Key 를 반환합니다.
- **CreateInnerBucket** : InnerBucket 을 생성하여 반환합니다.
- **CreateOuterBucket** : OuterBucket 을 생성하여 반환합니다.
- **GetWeekNo** : 대상 MoPlan 의 주차 정보를 반환합니다. 주차를 관리하는 기준이 적용 사이트에 따라 달라짐으로 이를 반영합니다.
- **SaveMoPlans** : 최종 Smoothing 된 결과로 수정된 MoPlan 을 업데이트합니다.

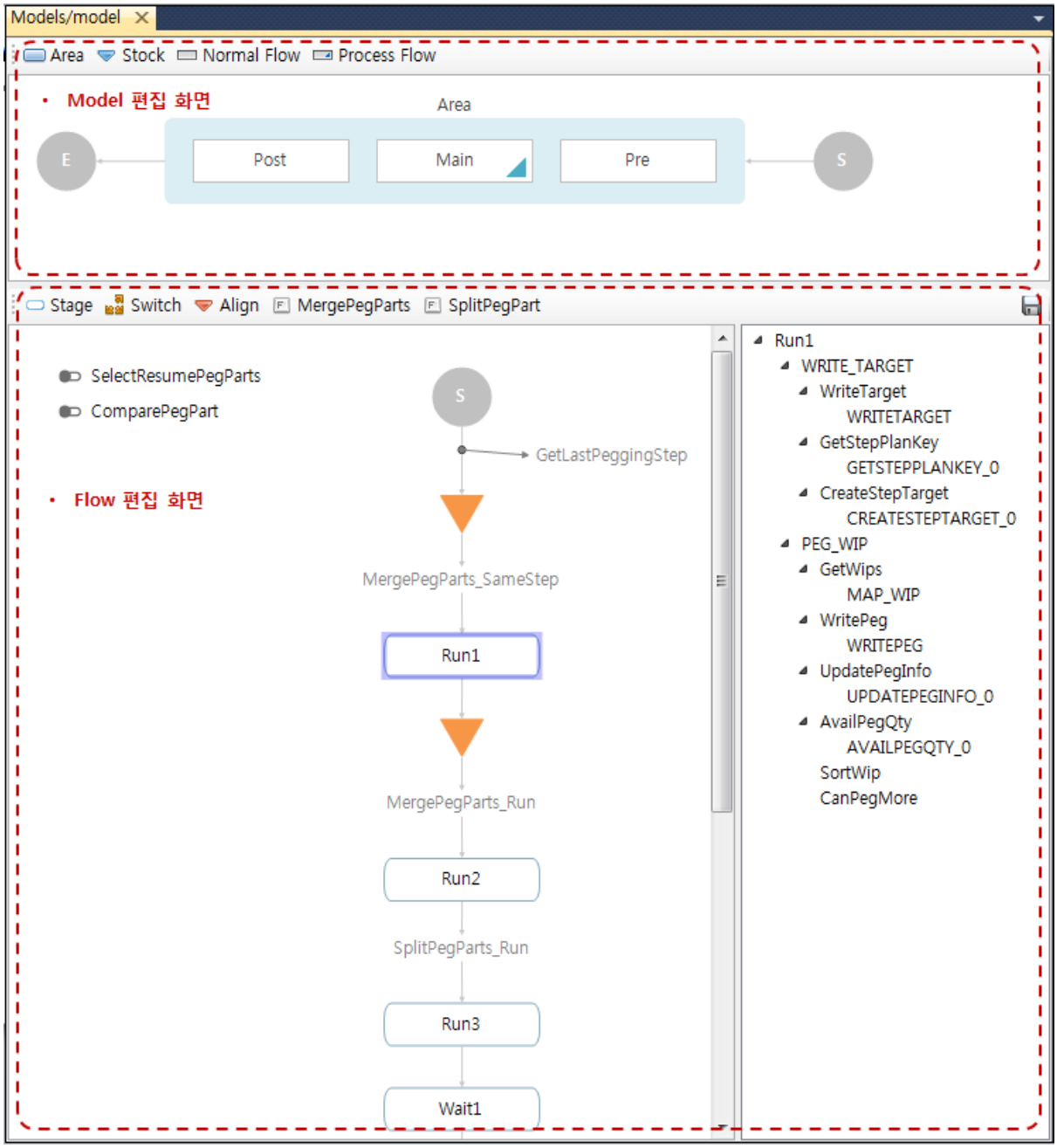
Pegging Modeler

개요

MOZART Pegging Library 는 Mozart Rule Flow 엔진을 기반으로 Pegging Logic 개발에 적합한 형태의 도구집합을 제공합니다. Pegging Logic 개발 도구 중 Diagram 을 그리는 방식으로 Pegging 수행에 필요한 Model 을 작성하는 기능이 Pegging Modeler 입니다. Pegging Modeler 는 기존에 Pegging Module 로 Logic 을 개발했을 때에 알기 어려웠던 Action 의 관계나 호출 순서 등을 Diagram 으로 나타내줌으로써 사용자가 좀 더 쉽게 모델링 할 수있도록 도움을 줍니다. 실행되어야 할 Action 의 순서가 명확한 Simulation 과는 달리 Pegger 는 실행될 Stage 가 정해져있지 않습니다. 따라서 Pegging Modeler 를 통해 Stage 순서를 정하여 Pegging Logic 을 개발할 수 있습니다. Pegging Modeler 의 기능은 Area 와 Stock, 그리고 Area 에서 실행할 Flow 의 순서와 관계를 정의하는 Model 편집 기능, Flow 에서 실행할 Stage 와 Function 사이의 순서와 관계를 정의하는Flow 편집 기능, 그리고 개발된 Model 을 실행하는 기능으로 구분됩니다.

- **Model 편집 기능** : Pegging Model 의 기본적인 흐름을 정의하는 역할을 하며, Area 와 Stock, 그리고 Area 에서 실행할 Flow 의 순서와 관계를 정의하는 기능을 제공합니다.
- **Flow 편집 기능** : Flow 에서 실행할 Stage 와 Function 사이의 순서와 관계를 정의하는 기능을 제공합니다. Stage, Switch, Align 등의 여러가지 Function 들로 Pegging Logic 을 표현할 수 있습니다.
- **Model 실행 설정 기능** : Model 및 Flow 편집 기능으로 개발한 Pegging Model 의 실행 설정사항을 변경하는 기능을 제공합니다. Pegging Model 이 두 개 이상이면 실행 순서를 변경할 수도 있습니다.

Pegging Modeler 의 편집 화면은 다음과 같이 구성됩니다. Model 편집 화면에서는 Area 와 Stock, 그리고 Area 에서 실행할 Flow 의 순서와 관계를 정의할 수 있고, Flow 편집 화면에서는 Flow 에서 실행할 Stage 와 Function 사이의 순서와 관계를 정의할 수 있습니다. 각 편집 화면에서 Item 은 상단의 툴바에서 Drag & Drop 하여 추가할 수 있습니다.



Model 편집

Pegging Modeler 에는 Pegging Model 에 Area 와 Stock, 그리고 Area 에서 실행할 Flow 의 순서와 관계를 정의하는 model 편집 기능과 Flow 에서 실행할 Stage 와 Function 사이의 순서와 관계를 정의하는 Flow 편집 기능이 있습니다. 그 중 model 편집 기능은 Pegging Model 의 기본적인 흐름을 정의하는 역할을 합니다. 본 장에서는 model 편집에 필요한 구성요

소(Item)와 이 구성요소(Item)를 이용하여 어떻게 Model 흐름을 정의하는지에 대해 설명합니다.

---

model 편집 구성요소(Item)

1. Default Node

model 편집 화면에서 Default 로 제공되는 Block 입니다. ‘S’ Node 와 ‘E’ Node 사이에 Area, Stock 등을 추가할 수 있습니다. ‘S(Start)’와 ‘E(End)’ 의 흐름이 뒤에서부터 시작하는 이유는 Pegging 이 BackWard Planning 을 기반으로 하기 때문입니다.

2. Area

Model 의 최상위 Level 에 추가 가능한 Block 입니다. Area 는 Pegging Model 에서 Flow(Normal Flow 및 Process Flow) 를 묶어주는 역할을 하며, 한 개 이상의 Flow 로 구성됩니다. Area 안에 Flow 를 추가하면 ‘S’ Node 에 가까운 Flow 부터 순서대로 실행됩니다. 이때, Flow 는 Area 에만 추가가 가능합니다.

3. Stock

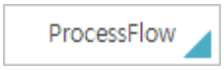
Model 의 최상위 Level 에 추가 가능한 Block 입니다. Stock 은 단순하게 Stage 들을 차례대로 실행시키는 기능을 하며, 하위에 Flow 를 추가할 수 없습니다.

4. Nomal Flow

Flow 중 하나이며, Area 하위에만 추가 가능한 Block 입니다. Normal Flow 는 Stock 과 유사하게 Stage 를 차례대로 실행시키는 기능을 합니다. Normal Flow 에는 Stage 만 추

가 가능합니다.

5. Process Flow



Flow 중 하나이며, Area 하위에만 추가 가능한 Block 입니다. Process Flow 는 실제로 Pegging 을 실행하는 Flow 입니다. Process Flow 에는 Stage 이외에 Switch, Align 등 여러가지 Function 들을 추가할 수 있으며, 이 Function 들을 사용하여 Pegging Logic 을 표현할 수 있습니다.

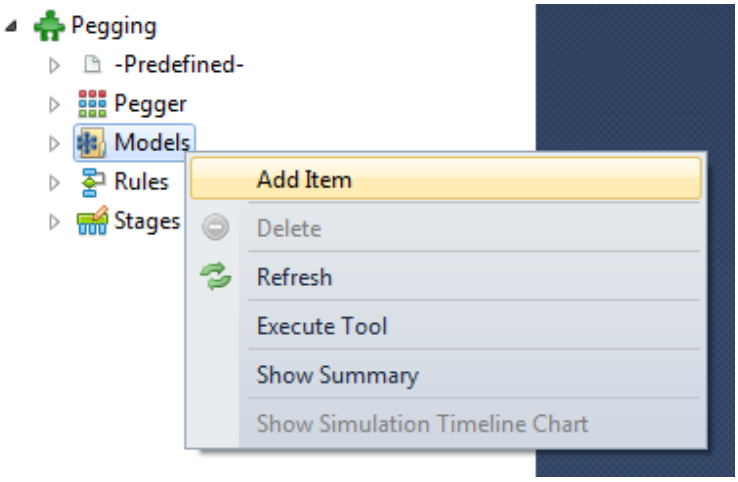
---

model 편집 방법

Pegging Model 은 기본적으로 Diagram 에 등록된 Area 와 Stock 을 실행하는 기능을 합니다. 다음은 model 편집 화면에 Area, Stock 등을 추가하여 Model 흐름을 정의하는 과정입니다.

‘Model’ Item Block 추가하기

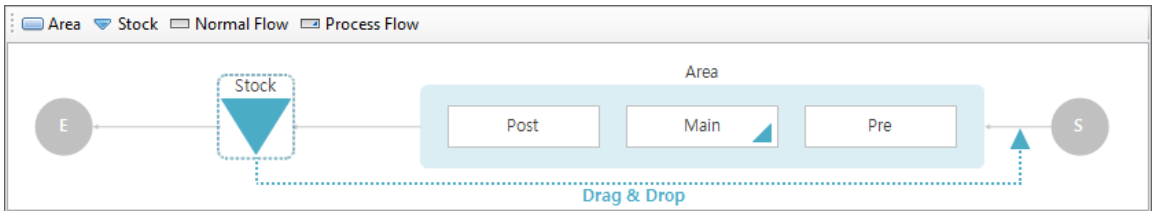
1. MOZART Explorer 의 **Pegging > Models** 노드를 선택합니다.
2. 팝업 메뉴의 **[Add Item]** 명령을 실행합니다.

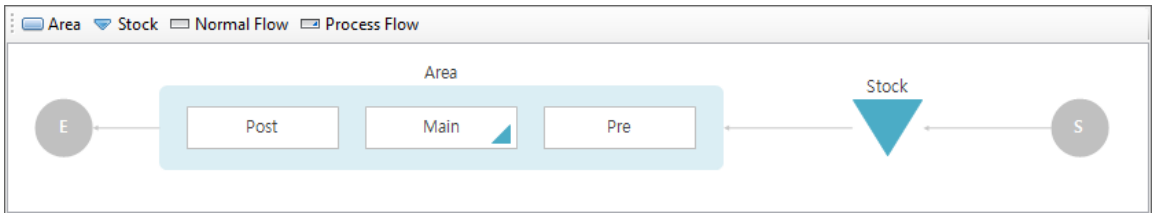


3. Create new pegging model Dialog 에서 ‘Model 명’을 입력하고 **[OK]** 버튼을 클릭합니다.
4. model 편집 화면이 활성화되면 편집 화면 상단의 툴바에서 Item Block 을 Drag & Drop 하여 Model 흐름을 정의합니다.

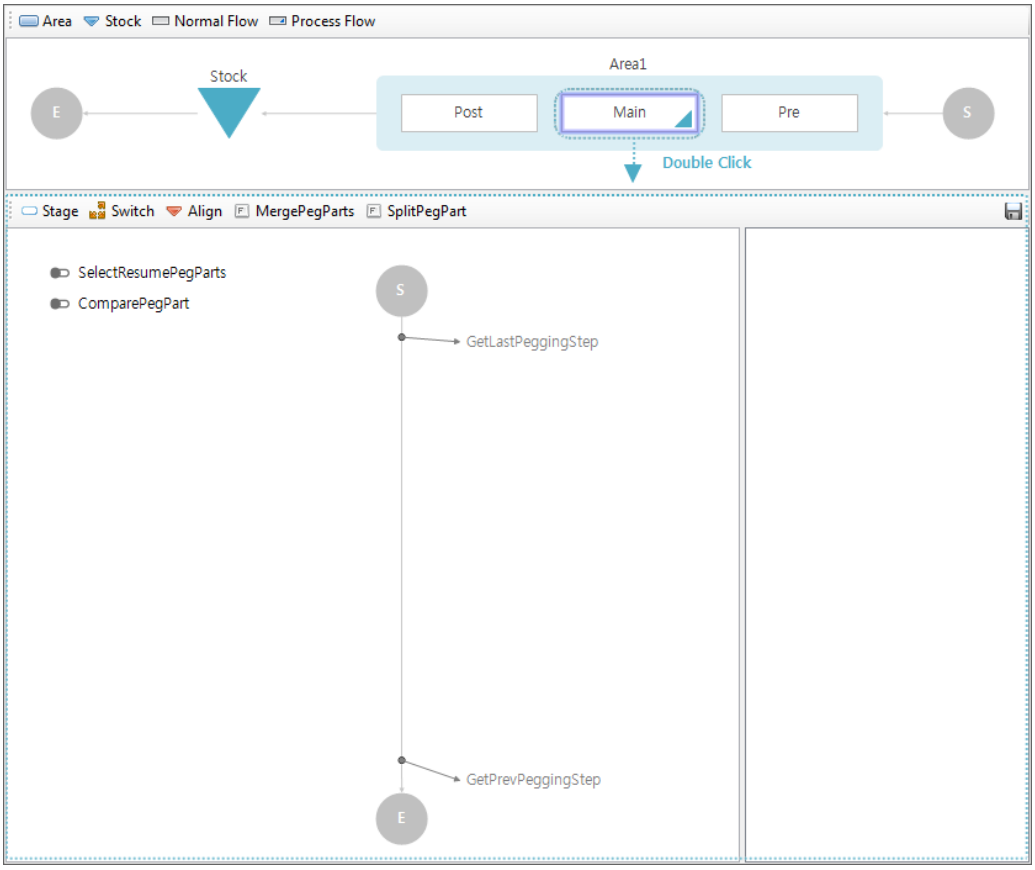
5. Item Block 을 Default Node 상에 Drag & Drop 하면 'Item Block 명'을 입력하는 창이 열립니다.
6. 입력창에 입력하고자 하는 'Item Block 명'을 입력하고 **[OK]** 버튼을 클릭합니다.
7. 그러면 다음과 같이 Item Block 이 추가됩니다. 'Stock', 'Normal Flow', 'Process Flow' Block 도 같은 방식으로 추가할 수 있습니다.

8. 만약 추가한 Item Block 의 위치를 변경하고 싶다면, 순서를 변경하고자 하는 Item Block 을 선택한 후 Drag & Drop 하여 위치를 변경할 수 있습니다





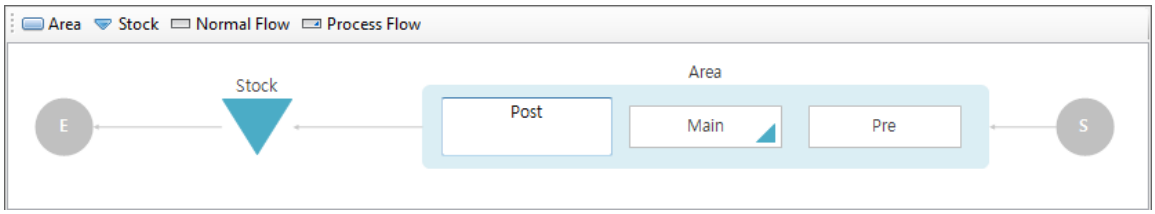
9. 일반적으로 Area 와 Stock 은 직렬로 연결하지만 병렬로 연결할 수도 있습니다. Area 와 Stock 이 직렬로 연결된 경우 **'S'** Node 에 가까운 것부터 순서대로 실행이 되며, 병렬로 연결된 경우 동일한 PegPart 로 실행한 뒤에 결과를 병합합니다.
10. 'Area' 를 제외한 'Stock', 'Normal Flow', 'Process Flow' Block 들을 더블 클릭하면 Flow 편집 화면이 열립니다. Flow 편집 화면에서는 실행할 Stage 와 Function 사이의 순서와 관계를 정의할 수 있습니다. 이에 대한 자세한 사항은 **Flow 편집**을 참조하시기 바랍니다.



‘Model’ Item Block 수정 및 삭제하기

등록한 Item Block 은 수정 및 삭제가 가능합니다. 다음은 Item Block 을 수정하고 삭제하는 과정입니다.

1. ‘Item Block 명’을 바꾸고자 하는 Item Block 을 선택하고 오른쪽 마우스 버튼을 한 번 클릭하면 이름을 변경할 수 있는 상태가 됩니다.
2. 이 상태에서 이름을 변경하고 Block 외부를 클릭한 후 **[모두 저장]** 버튼을 클릭하면 변경된 사항이 반영됩니다.



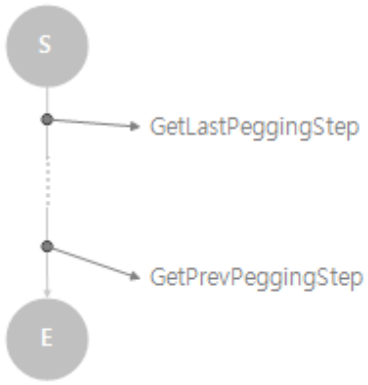
3. 등록한 Item Block 을 삭제하려면 삭제하고자 하는 Item Block 을 선택하고 Keyboard 의 **[Delete]** 버튼을 누르면 해당 Block 을 삭제할 수 있습니다.

Flow 편집

Flow 는 Normal Flow 와 Process Flow 로 구분됩니다. Normal Flow 는 Stock 과 유사하게 Stage 를 차례대로 실행시키는 기능을 합니다. Normal Flow 에서 PegPart 는 '**S**' Node 에서 출발하여 각 Stage 의 Rule 을 실행한 뒤에 '**E**' Node 를 끝으로 Flow 실행을 종료합니다. Process Flow 는 실제로 Pegging 을 실행하는 Flow 입니다. Stage 외에 Switch, Align 등 여러가지 Function 들로 Pegging Logic 을 표현할 수 있습니다. 본 장에서는 Flow 편집에 필요한 구성요소(Item)와 이 구성요소로 Pegging Logic 을 표현하는 과정을 설명합니다.

Flow 편집 구성요소(Item)

1. Default Node



   Flow 편집 화면에서 Default 로 제공되는 Block 및 Definition 입니다. Definition 중 GetLastPeggingStep 은 PegPart 가 Process Flow 에 진입했을 때 최초로 실행됩니다. GetLastPeggingStep 의 반환값은 PegPart 의 CurrentStep Property 에 저장됩니다. GetPrevPeggingStep 은 한 Step 에 대한 모든 Stage 가 실행된 다음에 이전 Step 으로 넘어갈 때 호출됩니다. PegPart 의 CurrentStep 이 이전 Step 으로 변경되면 PegPart 는 Process Flow 의 GetLastPeggingStep 다음 Node 부터 다시 실행됩니다.

2. Stage

   Stock, Normal Flow, Process Flow 모두에 추가 가능한 Block 입니다. Stage 는 Rule Flow 엔진의 Model 구성 주요 요소로서 로직 실행과 전체 Flow 구성의 기본 단위입니다. Pegging Modeler 에서 Flow 에 Stage 를 추가하려면 Pegging Module 하위에 Stage Item 을 추가해야 합니다. 이에 대한 자세한 사항은 Stage 편집을 참조하시기 바랍니다.

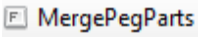
3. Switch

Process Flow 하위에만 추가 가능한 Block 입니다. Switch 는 조건에 따라 PegPart 의 흐름을 조정할 수 있는 기능을 합니다. Switch Expression 의 반환값과 Switch Case 의 값을 비교해서 PegPart 가 흘러갑니다.

4. Align

Process Flow 하위에만 추가 가능한 Block 입니다. Align 은 특정 조건에 해당하는 PegPart 를 실행 대상 PegPart 에서 잠시 제외하는 기능을 합니다. 특정 Stage 의 실행 우선순위가 낮아서 다른 PegPart 와 함께 Merge 되어야 하는 경우 Align 에서 처리합니다.

5. MergePegParts

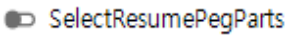


Process Flow 하위에만 추가 가능한 Definition 입니다. MergePegParts 는 Align 에 의해 제외된 PegPart 들이 다시 실행 대상으로 복귀했을 때 Merge 해서 다음 로직을 실행해야 하는 경우 사용됩니다. MergPegParts 는 Align 다음에만 위치할 수 있습니다.

6. SplitPegPart

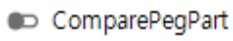
Process Flow 하위에만 추가 가능한 Definition 입니다. SplitPegPart 는 하나의 PegPart 를 두 개 이상의 PegPart 로 분할할 때 사용하며, GetLastPeggingStep 과 GetPrevPeggingStep 사이에서는 사용 위치에 제한이 없습니다.

7. SelectResumePegParts



Process Flow 하위에만 추가 가능한 Definition 입니다. SelectResumePegParts 는 모든 PegPart 가 실행 대상에서 제외되어 더 이상 실행 대상 PegPart 가 존재하지 않는 경우 다시 실행 대상으로 복귀할 PegPart 를 선택할 때 호출됩니다. Align 이름과 Align 로직에서 설정한 Key 값을 참조해서 복귀할 PegPart 를 선택할 수 있습니다.

8. ComparePegPart

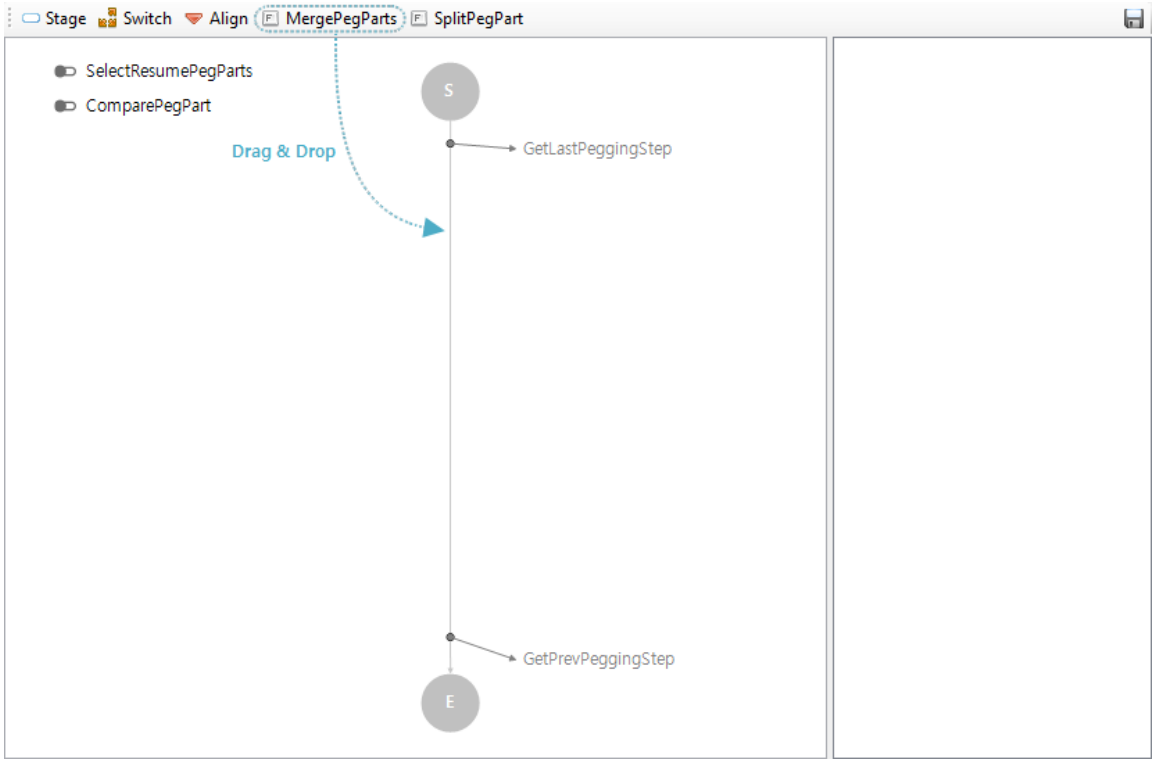


Process Flow 하위에만 추가 가능한 Definition 입니다. ComparePegPart 는 Flow 상의 여러 개의 PegPart 를 비교하여 실행 우선순위를 결정하는 역할을 합니다. 우선 순위를 결정하는 방식은 PegPart ‘x’ 와 PegPart ‘y’ 를 비교하여 반환되는 int 값이 음수(-)이면 PegPart ‘x’ 의 우선순위가 높아지고, 양수(+)이면 PegPart ‘y’ 의 우선순위가 높아집니다. 만약 반환값이 ‘0’ 이면 우선순위는 같게 됩니다.

Flow 편집 방법

‘Flow’ Item Block 추가하기

1. Model 편집 화면에서 등록한 ‘Model’ Item Block 을 더블 클릭합니다.
2. Flow 편집 화면이 활성화되면 편집 화면 상단의 툴바에서 Item Block 을 Drag & Drop 하여 Flow 를 정의합니다.



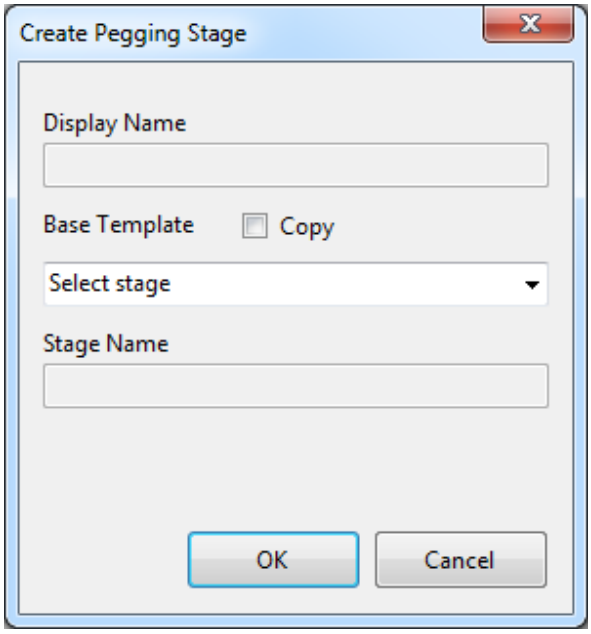
3. Item Block 을 Default Node 상에 Drag & Drop 하면 ‘Item Block 명’을 입력하는 창이 열립니다.

4. 입력창에 입력하고자 하는 'Item Block 명'을 입력하고 **[OK]** 버튼을 클릭합니다.
5. 그러면 Default Node 상에 Item Block 이 추가됩니다.

> **Note**
> **'Stage' Item Block 추가**

Pegging Modeler 에서는 Pegging Module 에서 미리 정의해놓은 Stage 를 사용합니다. (Stage 편집 참조) 따라서 Flow 편집화면에 'Stage' Item Block 을 추가할 때에는 Pegging Module 에서 정의해놓은 Stage 를 선택하는 것으로 Block 을 추가할 수 있습니다.

1. 'Stage' Item Block 을 Default Node 상에 Drag & Drop 하면 Create Pegging Stage 팝업창이 열립니다.
2. 팝업창의 'Select Stage' Combo Box 에서 등록해둔 Stage 를 선택합니다.



- **Display Name** : Pegging Modeler 의 Flow 편집 화면에 보여지는 Stage 의 이름입니다. Base Template Field 에서 Stage 를 선택하면 활성화됩니다.
- **Base Template** : Pegging Module 에서 정의한 Stages 목록입니다. Pegging Modeler 에서 'Stage' Item Block 을 등록하기 위한 'Base' 가 됩니다.
- **Stage Name** : Copy 한 Stage 의 이름입니다. Base Template 'Copy' 에 체크를 하면 활성화됩니다. 선택된 Stage 를 Copy 한 경우 해당 Stage 에서 지정한 모든 Rule

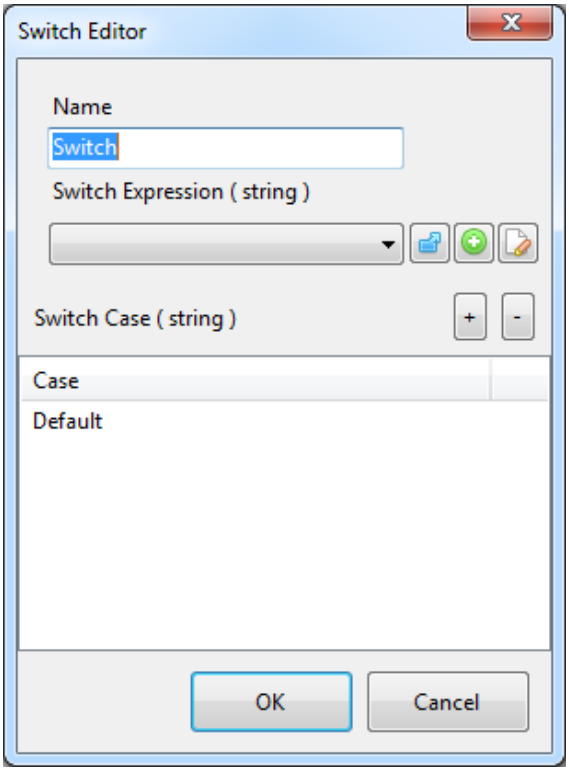
및 Property 정보가 Copy 되며, **Pegging Module > Stages** 하위노드에 Stage Name 에서 입력한 이름으로 신규 Stage 가 추가됩니다.

3. Stage 를 선택한 후 **[OK]** 버튼을 클릭하여 'Stage' Item Block 추가를 마무리합니다.
4. 추가된 'Stage' Item Block 을 더블 클릭하면 **Pegging Module > Stages** 노드에서와 마찬가지로 Stage Editor 팝업창이 열립니다. 팝업창에서 Rule 및 Property 를 편집할 수 있습니다.

> **Note**
> **'Switch' Item Block 추가**

Switch 는 Switch Expression 과 Switch Case 에 따라 PegPart 의 흐름을 조정할 수 있는 기능을 합니다. 따라서 'Switch' Item Block 을 추가할 때에 Switch Expression 과 Switch Case 를 정의해야 합니다.

1. 'Switch' Item Block 을 Default Node 상에 Drag & Drop 하여 'Item Block 명'을 입력한 후 [OK] 버튼을 클릭합니다.
2. 그러면 Switch Editor 팝업창이 열리고 팝업창에 Switch Expression 과 Switch Case 를 정의합니다.



▶️ Switch Expression 정의

- **Expression 선택** : 사전에 등록된 Load Expression 을 Combo Box 에서 선택할 수 있습니다.
- **Expression 추가** : PegPart 의 흐름을 조정할 수 있는 Switch 함수명과 빈 함수의 소스를 생성합니다.
  ➕ 버튼을 사용합니다. 함수가 생성되는 파일 명은 Main.cs 이며, Logic > Pegging 폴더 하위에 생성됩니다.
- **Expression 편집** : Combo Box 에 선택된 Expression 을 편집하기 위해 소스로 이동합니다. [버튼이름] 버튼을 사용합니다.
- **Expression name 편집** : Expression 의 함수 명을 편집하기 위해 사용합니다. [버튼이름] 버튼을 사용합니다.

▶️ Switch case 정의

- **Case 추가** : 신규 Case 를 추가합니다. ➕ 버튼을 사용합니다.
- **Case 삭제** : 선택한 Case 를 삭제합니다. Default Case 는 삭제할 수 없습니다. ➖ 버튼을 사용합니다.

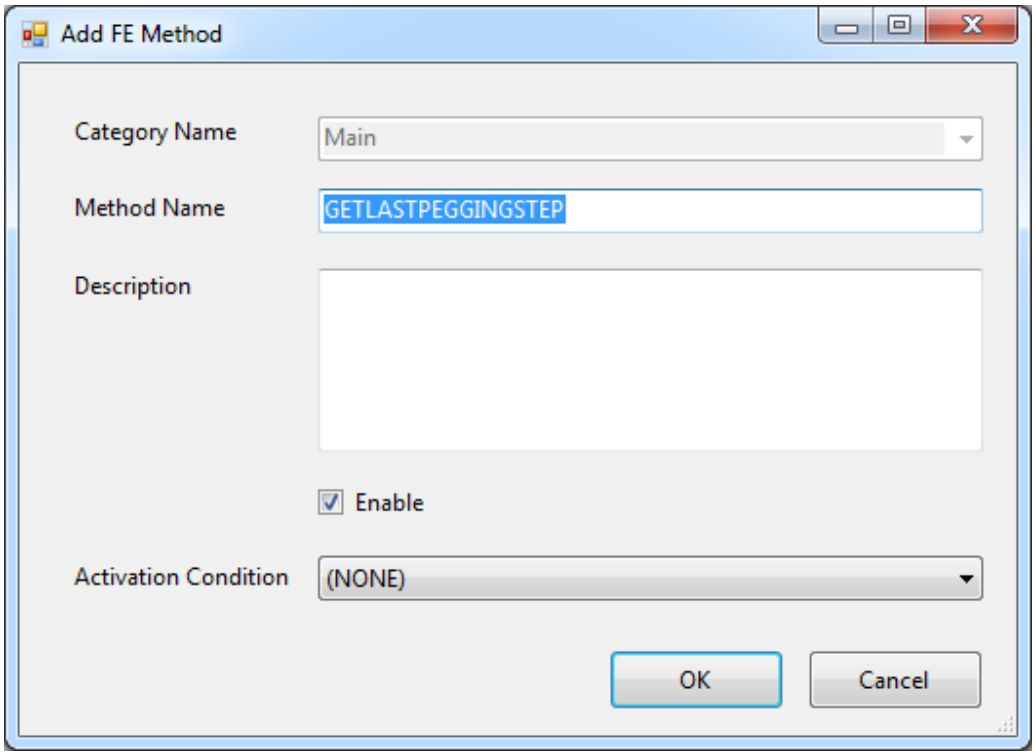
1. Switch Expression 과 Switch Case 편집을 완료하였으면 [OK] 버튼을 클릭하여 'Switch' Item Block 추가를 마무리합니다.

2. 추가된 ‘Switch’ Item Block 을 더블 클릭하면 Switch Editor 팝업창이 열리며 Switch Expression 과 Switch Case 에 수정사항이 있을 때 언제든지 편집이 가능합니다.

‘Flow’ Definition 추가하기

Pegging Model 에서 PegPart 의 흐름을 제어하는 Block 에는 Definition 을 추가할 수 있습니다. ‘Flow’ Item Block 중 Definition 을 추가할 수 있는 Block 은 ‘GetLastPeggingStep’, ‘GetPrevPeggingStep’, ‘Align’, ‘MergePegParts’, ‘SplitPegPart’, ‘SelectResumePegParts’, ‘ComparePegPart’ 입니다. 다음은 해당 Item Block 에 Definition 을 추가하는 과정입니다.

1. Definition 을 추가할 수 있는 Item Block 을 더블 클릭하여 Add FE Method 팝업창을 엽니다.
2. 팝업창에 ‘Method 명’과 Method 설명 등을 입력한 후 **[OK]** 버튼을 클릭합니다.



3. 그러면 Main.cs 소스 편집창에 다음과 같이 함수 원형이 생성됩니다.

```
public partial class Main
 {
     /// <summary>
     ///
     /// <param name="pegPart"/>
     /// <returns/>
```

```
    public Step GETLASTPEGGINGSTEP(Mozart.SeePlan.Pegging.PegPart pegPart)
    {
        return default(Step);
    }
}
```

4. 생성된 함수 원형 밑에 함수 본문을 작성하여 Pegging Logic 을 구현합니다.

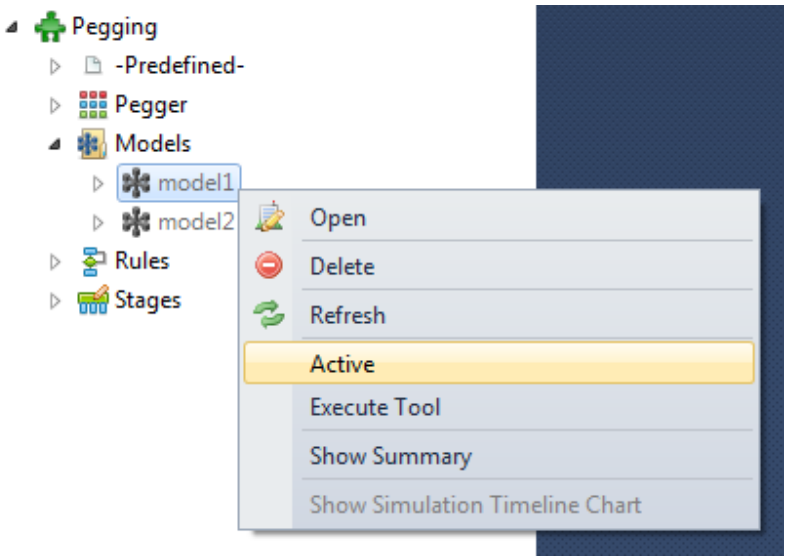
   Definition 구현 및 사용예

```
public Step GETLASTPEGGINGSTEP(Mozart.SeePlan.Pegging.PegPart pegPart)
{
   var pp = (SimpleMfgSemiconPegPart)pegPart;
   var proc = (SimpleMfgSemiconProcess)pp.Product.Process;
   var last = proc.LastStep;
   return last;
}
```

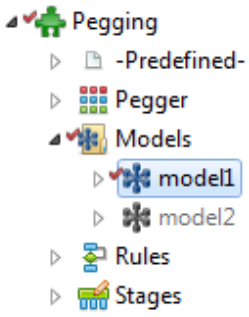
Model 실행

Pegging Modeler 에 Pegging Model 을 등록하면 기본적으로 비활성화된 상태로 등록됩니다. Pegging Model 을 실행하기 위해서는 실행 여부를 선택해야 합니다. 또한 등록된 Pegging Model 이 두 개 이상이면 실행 순서를 변경할 수도 있습니다.

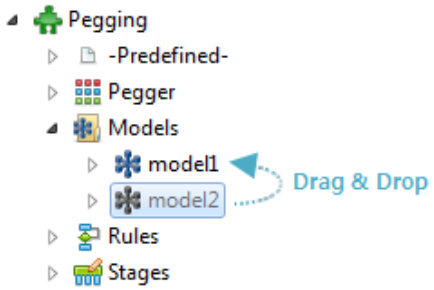
1. MOZART Explorer 의 Pegging Model 에서 오른쪽 마우스 버튼을 통해 **[Active]** 에 체크를 하여 Model 을 활성화시킵니다.

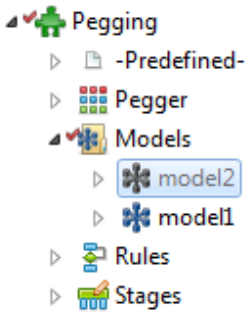


2. 활성화된 Pegging Model 의 아이콘은 파란색으로 변경되고, 해당 Pegging Model 은 실행됩니다.



3. **[Active]** 체크를 해제하는 경우 해당 Pegging Model 은 실행되지 않으며, 아이콘이 회색으로 변경되어 비활성화 상태가 됩니다. 만약 모든 Pegging Model 이 비활성화된 경우 기존의 Pegger 에서 정의한 Pegging Logic 이 실행됩니다.
4. Pegging Model 은 MOZART Explorer 에 등록된 Node 순서대로 실행됩니다. 이때, 등록된 Pegging Model 이 두 개 이상일 경우 실행 순서를 변경할 수 있습니다. 순서를 변경하고자 하는 Pegging Model 을 선택한 상태에서 Drag & Drop 하여 실행 순서를 변경합니다.

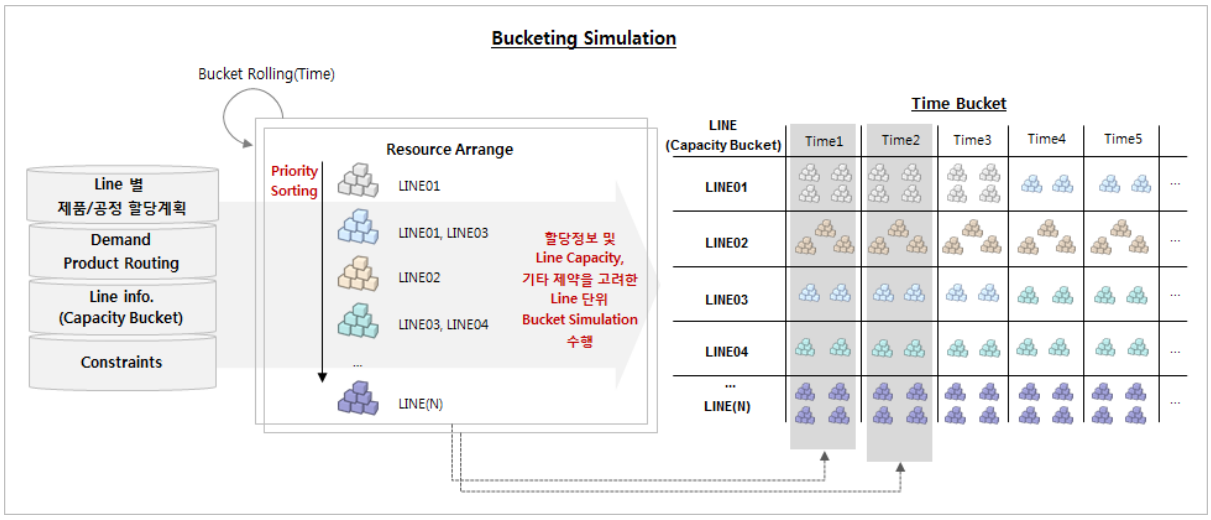




CBS MODULE

Capacity Bucket Simulation 개념 및 개요

Capacity Bucketing Simulation 은 생산자원을 capacity bucket 으로 모델링하고 Simulation 의 시간진행을 Bucket Rolling 주기에 맞도록 진행시켜가면서 주어진 Demand (PO:purchase order, WO:work order)의 우선순위와 제약을 고려하여 capacity bucket 에 할당하는 방식의 시뮬레이션 엔진입니다.



CBS 의 경우 장비단위의 로딩 시뮬레이션보다 심플하게 모델링이 가능한 장점을 가지고 있습니다. 라인을 한개 혹은 몇개의 주요 자원집합으로 모델링하고 해당 자원집합에 Demand 를 시간 순서에 따라 배치를 하는 방식으로 장비단위의 스케줄보다는 좀더 상위 레벨의 투입계획이나 생산계획을 수립할때 사용하기 적당한 방식입니다. 또한, CBS 엔진은 Bucketing 을 통한 Demand-자원 배정의 Heuristic 한 로직 개발방식과 Simulation 기법을 통합하여 시간의 흐름에 따라 Demand 와 Capacity Bucket, 제약의 속성을 동적으로 변경하면서 Bucketing 을 할 수 있음으로 기존의 Heuristic 방식만을 사용한 Planning 솔루션에 비해 모델링의 확장성이 높다는 장점이 있습니다.

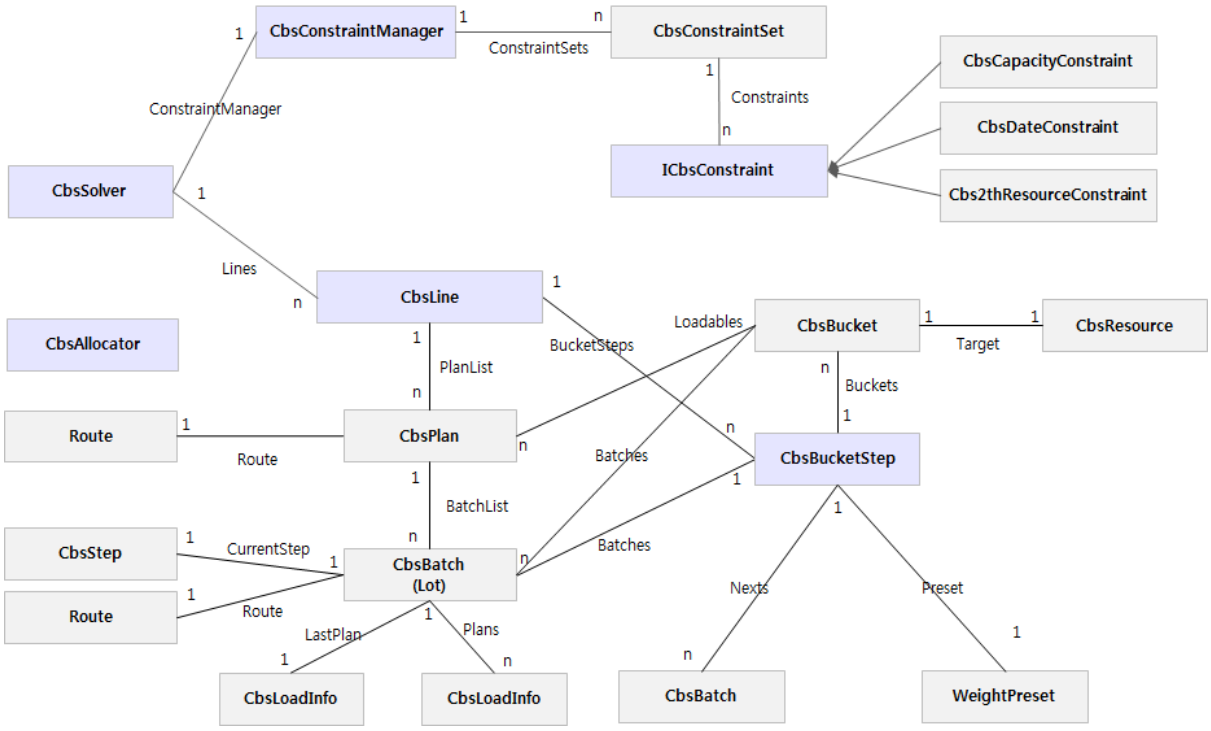
CBS 모델의 적용 가능 예

1. **투입계획 수립모델**
    - 문제 정의 : 라인별 제품의 일별 투입량을 결정해주어야 하는 문제
    - 제약 :
        - 각 제품별로는 기간별 투입 한계수량 제약이 있음
        - 특정 제품의 경우 투입시 별도의 자재가 있어야 생산이 가능한 경우가 존재함
        - 특정 제품의 경우 생산이 불가한 기간을 지정할 수 있음

2. **Bucketing 을 통한 생산 스케줄 수립 모델** (Simple 한 Loading Simulation 과 같은 방식으로 모델링 가능)
    - 문제 정의 : 생산제품이 다수의 공정을 거쳐 생산되며, 생산공정별로 서로 다른 장비(Resource)그룹에서 작업수행이 되는 공장에서 장비그룹별 생산계획을 수립하는 문제
    - 제약 :
        - 각 제품별/공정에 기간별 생산한계수량 제약이 있을수 있음
        - 제품의 특정 공정은 처리가 가능한 장비그룹정보가 없이 진행할 수 있음
        - 제품별/공정별로 별도의 자재가 있어야 생산이 가능한 경우가 존재함
        - 제품/공정별 생산 불가 기간을 지정할 수 있음
        - 특정 공정 처리 전/후 작업물이 Split 될 수 있음

CBS 모듈 클래스 설명

CBS 모듈을 사용하기 위한 클래스 구조는 아래의 그림과 같습니다.



- **CbsSolver**

  CBS 모듈의 Main Class 입니다. 시뮬레이션 시작부터 종료될때 까지의 전체 로직 제어를 담당하며 생산라인에 대한 모델정보와 제약정보를 속성으로 가집니다.

- **CbsAllocator**

  Bucket Rolling 시에 생성되어 제품의 Allocation 전체를 제어하는 클래스입니다. Allocator 는 Rolling 시점마다 생성이 되며 해당 시점의 정보로 초기화 됩니다.

CbsAllocator 의 클래스 설명

- **CbsConstraintManager**

  Allocation 시에 사용될 각종 제약정보를 관리하는 클래스입니다. Constraint 의 정보를 기반으로 초기화 및 Rolling 시점에 상태변경관리를 하거나 ConstraintSet 을 cache 에 저장하여 검색속도를 개선하는 역할을 수행합니다.

- **CbsConstraintSet**

  동일한 제품유형에 적용되거나 유사한 특성을 가진 Constraint 의 묶음 클래스입니다.

- **ICbsConstraint**

  제약 정보를 처리하기 위한 Interface 입니다.

- **CbsCapacityConstraint**

  일반적인 제약에 대한 기본 클래스로 주로 최대 생산가능 조건에 대한 제약입니다. 예를 들어 특정 제품이나 제품그룹의 제품을 Bucket Cycle Time 동안 정해진 제약수량 이상 생산/투입할 수 없도록 하는 제약 유형을 모델링할 때 사용할 수 있습니다.

- **CbsDateConstraint**

  특정 기간동안만 생산이 가능하나 해당 기간에 생산을 하지 못하도록 처리하기 위한 제약입니다.

- **Cbs2thResourceConstraint**

  해당 제품의 생산을 위해 추가적으로 필요한 자원에 대한 제약입니다.

- **CbsLine**

  하나의 공장에 대한 데이터 모델입니다. Bucketing 을 독립적으로 실행하기 위한 최상위 모델 클래스입니다. 초기화시에 CbsResource 의 Line 속성을 설정하면 자동으로 생성되며, InitControl 을 통해 초기화시 커스터마이징을 할 수 있습니다. 라인별 제품의 혼류 생산이 가능한 경우에는 실제 생산라인을 CbsLine 으로 모델링 하지 않고 Bucket 으로 처리

할 수 있습니다. CbsLine 은 한개 혹은 복수개의 CbsBucketStep 으로 구성될 수 있습니다.

- **CbsBucketStep**

  Bucket 을 구성하는 공정그룹에 대한 모델입니다. 생산자원인 Capacity Bucket 인 CbsBucket 정보를 가지고 Allocator 에 의해 Allocation 이 이루어지는 시점에 실제로 개별 CbsBucketStep 단위로 Allocation 이 수행됩니다. Allocation 시점에 Preset 을 사용하는 경우에는 WeightPreset 속성을 설정해야 하며 그렇지 않은 경우에는 Control 의 Batch Sorting Action 의 로직에 따라 대상 Batch 가 소팅됩니다.

- **CbsBucket**

  Capacity 를 정의할 수 있는 Bucket 데이터 클래스입니다. 물리적인 생산자원을 모델링하기 위한 클래스로 장비,장비그룹,생산라인 등을 하나의 Bucket 으로 모델링 할 수 있습니다. 모델링 방식은 문제의 종류에 따라 다르게 적용할 수 있습니다. CbsResource 정보에서 자동으로 생성됩니다.

- **CbsResource**

  CbsBucket 으로 구성하기 위한 자원의 정적 속성정보를 저장하고 CBS 모듈의 초기화를 위한 데이터모델입니다. 주요속성은 아래와 같습니다. 초기화시에 사용자가 직접 생성해주어야 합니다.

  - Line : Bucket 을 포함하는 라인(CbsLine) 의 명칭입니다. 초기화시에 CbsResource 리스트의 설정된 라인 이름으로 CbsLine 을 자동 생성합니다.
  - Name : Mapping 되는 CbsBucket 의 명칭입니다. CbsResource 에 설정된 Name 속성으로 CbsBucket 을 자동 생성합니다.
  - Group : Bucket Group 의 명칭입니다. BucketGroup 단위로 Allocation 이 수행됩니다. 실제로는 CbsBucketStep 과 Mapping 되는 속성입니다.
  - Step : CbsBucketStep 의 ID 로 사용됩니다. CbsBucketStep 은 Bucket 을 포함하는 BucketGroup 이 됩니다.

- **CbsPlan**

  Allocation 대상이 되는 Demand 정보로 보통 work order (W/O) 혹은 purchase order (P/O) 단위의 정보로 생성합니다. CBS 모듈의 기본 Input 정보 입니다. Demand 를 위해 Release 되는 Batch 를 포함합니다. 초기화시에 사용자가 직접 생성해주어야 합니다.

- **CbsBatch**

  CBS 모듈의 Entity Class 로 일반적으로 생산라인의 Lot 대응되는 시뮬레이션 모델입니다. 초기화시에 CbsPlan 을 생성할때 함께 생성합니다. 사용자가 생성해야 합니다. CbsPlan 에 기본적인 Batch 생성함수를 제공하며, 필요시 사용자 환경에 맞도록 생성해서 사용합니다.

- **CbsLoadInfo**

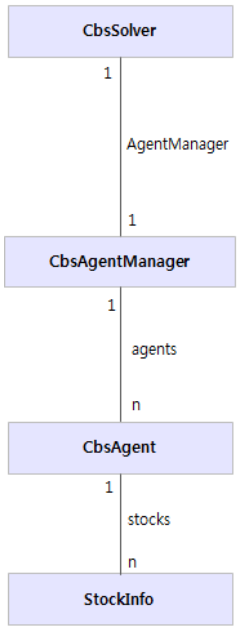
  작업물인 CbsBatch 의 작업계획 정보입니다. 가장 최근(마지막)의 계획정보는 LastPlan 속성으로 참조할 수 있습니다.

- **CbsStep**

  제품(Product)의 Routing 을 구성하는 공정 데이터모델입니다. Simulation Engine 의 Step 을 상속받아 구현된 모델로 BOP 를 구성하는 요소이며 Route 속성으로 해당 Route 정보를 참조할 수 있습니다. 초기화시점에 사용자가 생성해주어야 합니다.

- **Route**

  제품의 Routing 정보 클래스입니다. CbsStep 의 집합을 속성으로 가집니다. 제품 생산을 위해 여러 공정을 거치도록 모델링을 할 경우에는 Route 를 정보를 구성해야만 합니다. CbsBatch 는 Route 의 구성된 Step 순서에 따라 Step change 하면서 Simulation 을 수행합니다. 초기 CbsPlan 을 생성할 때 Route 정보를 설정하면 CbsBatch 는 해당 Plan 의 Route 속성을 가지도록 초기화 됩니다. 초기화시점에 Route 정보는 사용자가 생성해 주어야 합니다.



CbsAgentManager

속성	의미
Now	현재시각 (Time 타입)
NowDT	현재시각 (DateTime 타입)
Add(string id)	Id 에 해당하는 CbsAgent 를 생성하여 추가합니다.
Remove(string id)	Id 에 해당하는 CbsAgent 객체를 제거합니다.
GetAgent(string id)	Id 에 해당하는 CbsAgent 객체를 반환합니다.
AddBatch(batch)	**Batch를 대상이 되는 Agent 에 추가합니다.**
RemoveBatch(batch)	Batch를 대상이 되는 Agent 에서 제거합니다.

CbsAgent

속성	의미
Stocks	Agent 에 저장된 재고(재공) 정보. Dictionary<object, StockInfo>
Id	Agent ID
Priority	Agent 실행 순서
Enter(batch)	Agent 에 작업물 저장(Stock 에 저장되며, key 에 따라 다른 Stock 에 저장)
Remove(batch)	Agent 에 저장된 작업물을 제거(Agent 에서 Out 처리함)

StockInfo

속성	의미
Key	재고의 ID 정보 객체 (object)
Batches	재고ID 에 해당하는 실제 Batch 객체 집합(SearchQueueEx<CbsBatch>)

- **CbsAgentManager**

  Cbs모듈에서 사용되는 CbsAgent를 관리하기 위한 클래스입니다. 사용자가 Agent 를 사용할때는 Solver 의 AgentManager를 통해 배치를 등록해서 사용할 수 있습니다.

- **CbsAgent**

  Rolling 시점별로 실행될 Agent 로직을 처리하기 위한 클래스입니다. 사용자가 추가한 Batch 들을 Agent 의 Stock 에 적재하고 있다가 Rolling 주기에 Stock 을 평가해서

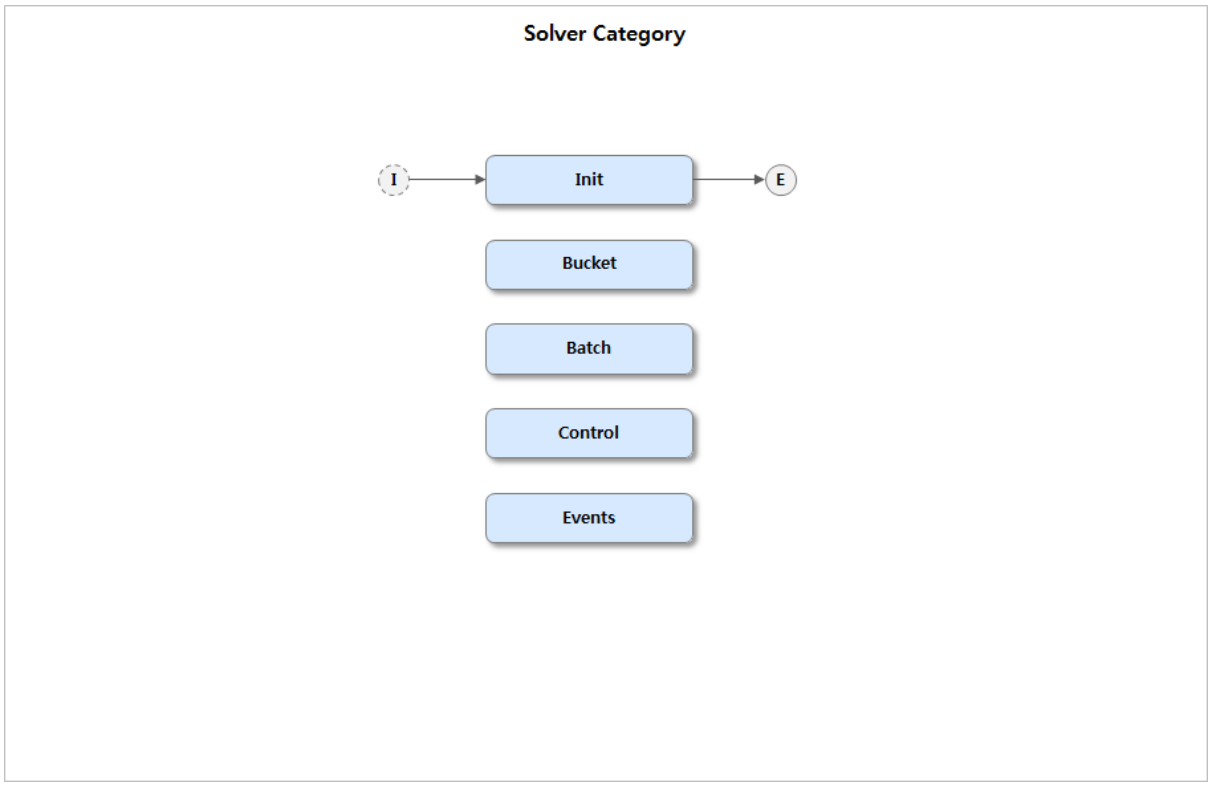
Release 할 Batch 를 결정하여 처리합니다. Agent 를 통해서 공정간 재고지점, 투입로직의 처리, 출하를 위한 로직등을 처리할 수 있습니다.

- **StockInfo**

  Agent 에 적재되어 있는 재공(Batch) 를 Key 별로 저장하기 위한 클래스입니다.

CBS Solver

Cbs 모듈의 초기화 및 주요 모델이 되는 Bucket, Batch 및 Allocator 의 주요 로직을 커스터마이징하기 위한 Control, Event 처리를 위한 FEComponent 의 집합입니다.



Init

Cbs 모듈을 구성하는 Resource(Bucket) 와 Batch(CbsPlan) 을 생성합니다. 또한 BucketGroup(BucketStep) 및 Bucket 등의 주요 객체의 초기화를 수행하며 Preset 정보를 생성하여 등록합니다.

Bucket

Bucket Rolling 및 Allocation 가능한 시간에 대한 제어를 수행하기 위한 FEComponent 입니다.

Batch

CbsBatch 및 BatchGroup을 제어하기 위한 FEComponent 입니다.

Control

Cbs Module 의 전체 제어를 수행하기 위한 FEComponent 입니다. Cbs 모듈에서 Allocation 을 실행하는 것은 Allocator 객체에서 수행하게 되는데 Allocator 객체의 초기화 시점부터 Rolling 이 시작, 종료까지의 모든 로직의 제어를 할 수 있도록 구성되어 있습니다.

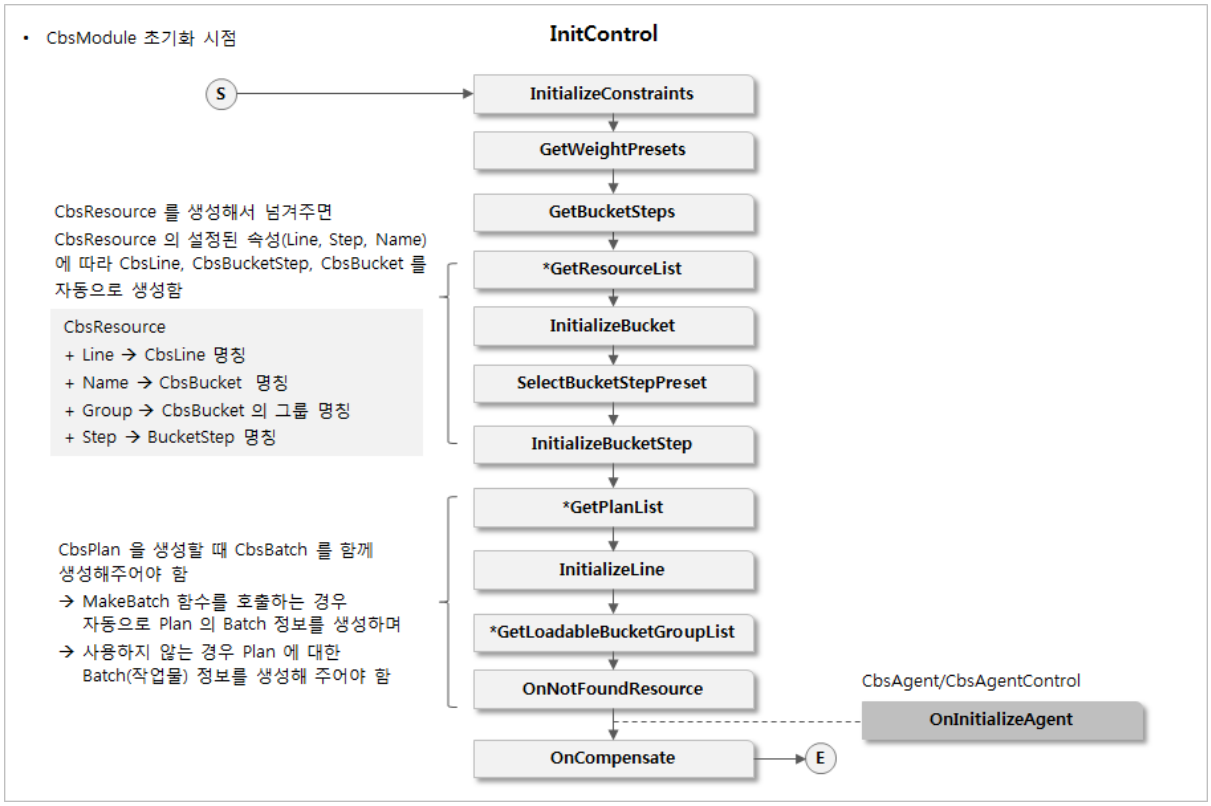
Events

Cbs Module 실행시 발생하는 Global 한 Event 시점(Day Change, Shift Change, ..)에 사용자 정의 코드를 작성할 수 있도록 FEAction 을 제공합니다.

Cbs init 컨트롤

CBS 모듈의 구동을 위한 초기화 작업을 수행하기 위한 함수 집합입니다

초기화를 위한 Init 컨트롤은 아래그림과 같은 순서로 실행됩니다. 각 함수의 역할에 따라 필요한 초기화 코드를 작성할 수 있습니다.



InitializeConstraints

Module 에서 사용될 Constraint(ICbsConstraint) 들을 CbsConstraintManager 에 등록합니다.

GetWeightPresets

Module 에서 Weight Preset 을 사용하여 Batch 우선순위를 평가하는 경우 사용될 Preset 정보를 만들어서 반환합니다.

GetBucketSteps

Capacity 자원의 집합인 BucketStep을 직접 등록할 때 사용합니다. BucketStep을 생성해서 반환하면 엔진에서 이를 등록합니다. 구현하지 않고 GetResourceList 호출 시 장비에 Group 명칭으로 등록하는 경우 자동 생성됩니다.

GetResourceList

필수 구현이 필요한 초기화 Action 입니다. CbsResource 를 생성해서 반환하면 CbsResource 의 설정 값에 따라 CbsLine, CbsBucket, CbsBucketStep 등의 모델 객체를 생성하고 초기화합니다.

InitializeBucket

CbsBucket 이 생성된 직후 사용자가 초기화 로직을 추가할 수 있는 FEAction 입니다.

SelectBucketStepPreset

BucketStep 의 Allocation 시 Batch Sorting 에 사용될 Preset 정보가 복수개인 경우 이중에 사용할 Preset 을 선택하도록 합니다. BucketStep 의 사용가능한 Preset 은 BucketStep 초기화 시점에 직접 할당된 Preset 혹은 Bucket(CbsResource) 에 할당된 Preset 입니다.

InitializeBucketStep

CbsBucketStep 이 생성된 직후 사용자가 초기화 로직을 추가할 수 있는 FEAction 입니다.

GetPlanList

필수 구현이 필요한 초기화 Action 입니다. Cbs 모듈의 Input 이 되는 Demand 리스트를 반환해야 합니다. Demand List 를 넘겨주면 라이브러리에서 CbsPlan 과 CbsBatch 를 생성하여 Allocation을 할 수 있도록 초기화합니다.

InitializeLine

CbsLine 의 추가적인 초기화 로직을 추가할 수 있는 FEAction 입니다.

GetLoadableBucketGroupList

필수 구현이 필요한 초기화 Action 입니다. 각 Plan 의 Route 상에서 Allocate 되어야 할 모든 BucketStep(BucketGroup)의 명칭 리스트를 반환하는 FEAction 입니다.

OnNotFoundResource

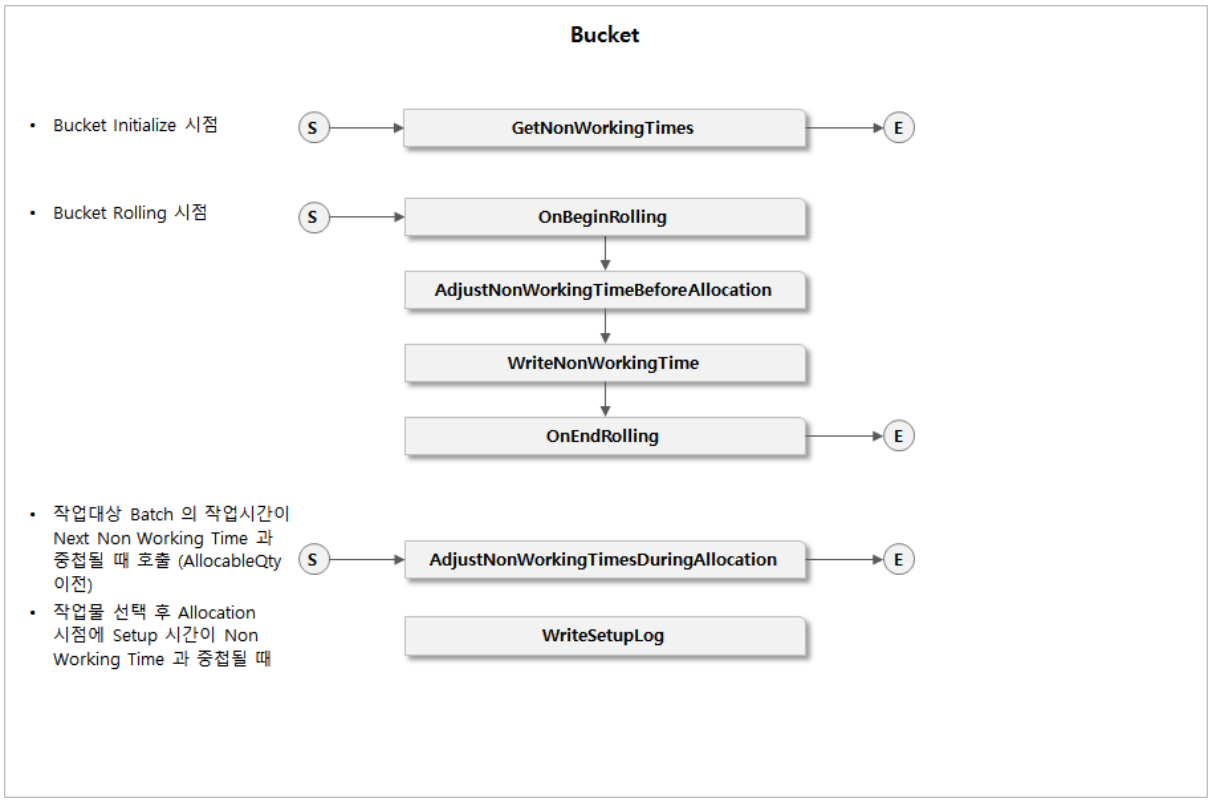
GetLoadableBucketGroupList 에서 반환된 BucketGroup 중 등록되지 않은 BucketGroup(자원그룹)이 있는 경우 호출됩니다. Argument 는 미등록된 BucketGroup 의 명칭입니다. 이 함수가 호출되는 경우에는 해당 BucketGroup을 사용하도록 설정된 Plan 의 경우 할당이 완료될 수 없습니다.

OnCompensate

시뮬레이션 초기 시점이 Bucket Rolling 시점과 일치하지 않는 경우 호출됩니다. 초기화시에 Constraint 별 잔여 Capacity 를 시간 비율에 따라 보정합니다. 예를 들어 Plan 시작 시점이 Rolling 구간의 80% 지난 시간일 경우 Capacity의 80% 를 사용한 것으로 자동으로 처리하고, 이 시점에 본 함수가 호출됩니다. 사용자는 Compensation 로직을 커스텀으로 수정할 수 있습니다. 모든 Constraint 의 정보를 가진 CbsConstraintManager 가 있으며, 현재시각, 차감 비율을 받아 사용자가 필요한 데이터를 업데이트합니다.

CbsBucket 컨트롤

Bucket Rolling 및 Allocation 가능한 시간에 대한 제어를 위한 FEComponent 입니다.



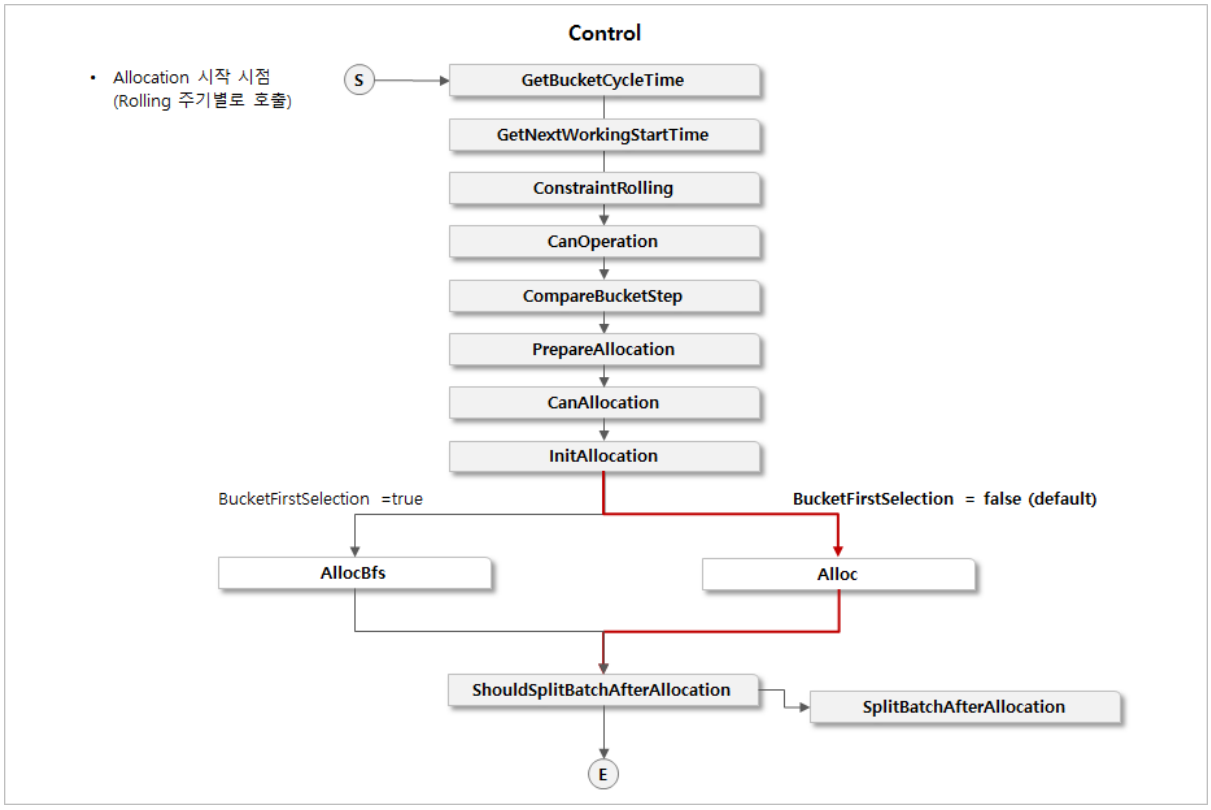
1. **GetNonWorkingTimes** : CbsBucket 의 초기화 시점에 Plan 전 주기동안 자원에서 제약으로 처리할 비가동시간 정보를 설정합니다.
2. **OnBeginRolling** : Bucket 에서 Rolling은 Bucket Capacity가 Reset되는 걸 의미합니다. OnBeginRolling 이벤트는 Reset 시작 시점에 호출됩니다.
3. **AdjustNonWorkingTimeBeforeAllocation** : 매번 Rolling 시점별로 모든 Bucket의 가용시간을 업데이트 합니다. 본 함수는 Bucket의 가용시간 업데이트 전 해당 Rolling 구간에서 반영되어야 할 비가동 시간을 설정하기 위해 사용합니다. 즉, 특정 Bucket에 대해 매일 Rolling을 한다고 할 때, 할당을 시작하기 전에 해당 일의 비가동시간을 사용자가 지정하여 반환합니다. CbsPeriodSection 리스트에 해당 주기의 값을 추가하는 방식으로 구현합니다.
4. **WriteNonWorkingTime** : NonWorkingTime 발생 시점에 호출되며 필요시 NonWorkingTime 에 대한 사유 등의 로그를 기록할 수 있습니다.
5. **OnEndRolling** : Bucket 의 Rolling 을 모두 마친 시점에 호출됩니다.
6. **AdjustNonWorkingTimesDuringAllocation** : Bucket 의 NonWorkingTime과 작업대상의 작업시간이 겹쳐지는 경우 호출됩니다. 사용자는 이 Action 을 통해 NonWorkingTime 과 Setup 및 작업시간이 중첩되는 경우 이를 Custom 로직으로 수정할 수 있습니다. 작업을 수행할 때 Setup 이 발생하면 Setup End 시각과 Non working time

이 겹칠 때 Setup 을 Non working time 에 포함시켜 처리하거나 Setup 시간에 Non working time 을 포함해서 처리하는 등의 작업을 수행할 수 있습니다.

7. **WriteSetupLog** : Setup이 발생되는 시점에 호출되며, Setup 로그를 기록할 수 있습니다.

Cbs Control 컨트롤

Cbs Module 의 모든 제어로직을 구현할 수 있도록 구성된 FEComponent 입니다.



1. **GetBucketCycleTime** : 다음 Rolling Event 시간을 설정합니다. Bucketing Rolling 주기를 동적으로 변경할 때 사용하며, 기본적으로는 Cbs Configuration 에서 설정된 Bucket Cycle Time 을 사용합니다.

   Default Code Sample

```
public Time GET_BUCKET_CYCLE_TIME_DEF(DateTime now, Time cycleTime, ref bool handle
d, Time prevReturnValue)
{
    return cycleTime;
}
```

2. **GetNextWorkingStartTime** : GetBucketCycleTime과 유사한 기능으로 다음 Rolling 시작지점을 지정할 수 있습니다.
3. **ConstraintRolling** : Rolling 이 시작될때 ConstraintManager 에 등록된 모든 Constraint 의 Capacity 등 설정값의 변경을 수행하기 위한 FEAction 입니다.

   Default Code Sample

```
public void CONSTRAINT_ROLLING_DEF(CbsCapacityConstraint cb, DateTime now, bool atBo
undary, bool atDayChanged, ref bool handled)
{
    if (cb is CbsDummyConstraint)
        return;

    if (atBoundary)
    {
        //cb.PreviousQty = 0;
        cb.CumulatedQty = 0;
        cb.MoveQty = 0;
    }
    else
    {
        cb.PreviousQty = cb.MoveQty;
        cb.CumulatedQty += cb.MoveQty;
        cb.MoveQty = 0;
    }
}
```

4. **CanOperation** : CbsLine 에 포함된 BucketStep 별로 대상 Allocation 주기에 Allocation 여부를 판별합니다. 기본값은 true 이며, false 값을 반환하는 경우 해당 BucketStep 이 해당 주기에 Allocation 을 하지 않습니다.
5. **CompareBucketStep** : 여러개의 Bucket Step 이 등록된 경우 Allocation 순서를 설정하기 위한 Comparer 를 구현합니다. 기본으로 CbsBucketStep 생성시에 설정된 Priority 의 오름차순에 따라 Allocation 을 실행합니다.

   Default Code Sample

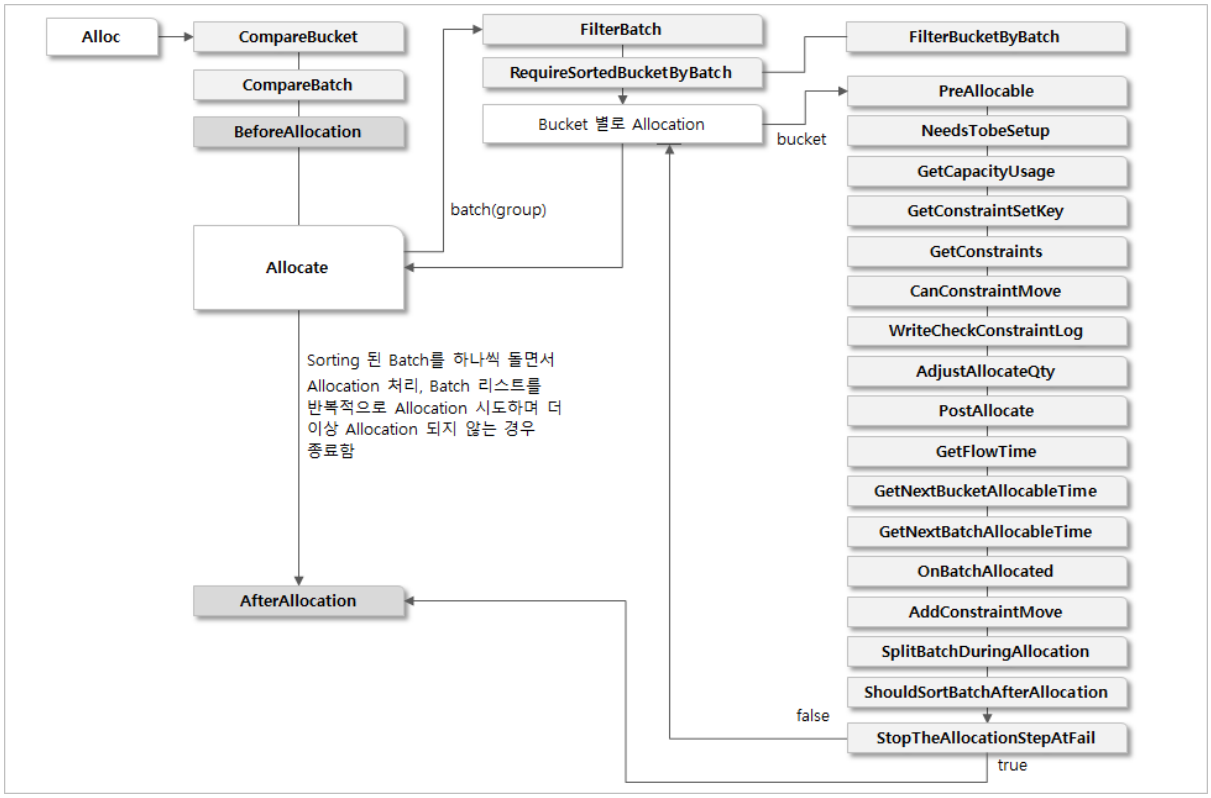
```
public int COMPARE_BUCKET_STEP_DEF(CbsBucketStep x, CbsBucketStep y, ref bool handle
d, int prevReturnValue)
{
   if (object.ReferenceEquals(x, y))
       return 0;

   int cmp = x.Priority.CompareTo(y.Priority);
   if (cmp == 0)
       cmp = x.Name.CompareTo(y.Name);
   return cmp;
}
```

6. **PrepareAllocation** : 특정 BucketStep 의 Allocation 이 시작되기 직전에 호출됩니다.
7. **CanAllocation** : 특정 Batch 가 특정 시점에 Allocation 가능 한지 여부를 판별합니다. Rolling 시점에 Allocator 에서 Allocate 대상 Batch 를 집계하는데 사용됩니다.
8. **InitAllocation** : Allocation 시작 전 초기화 수행이 필요한 로직을 개발하기 위해 사용되는 FEAction 입니다.
9. **ShoudSplitBatchAfterAllocation** : 모든 BucketStep 들의 Allocation 이 끝나고 남은 Batch (작업물)이 분리가 되어야 하는지 여부를 판별합니다. 특정 공정이 끝나고 작업물을 분리할 필요가 있는 경우 이 Action 을 사용하여 로직을 구현합니다.
10. **SplitBatchAfterAllocation** : ShoudSplitBatchAferAllocation 의 값이 true 인 기존의 Batch 의 속성을 변경하고 분리된 신규 Batch 를 생성해서 반환하도록 로직을 구현합니다.

Main Control logic 에서 CbsBucketStep 이 결정되고 해당 Step을 처리해야 하는 Batch 가 추출된 상태에서 아래의 두가지 Routine 에 따라 선별적으로 작업물을 할당 할 수 있습니다. CbsBucketStep.BucketFirstSelection 속성 값에 따라 실행 방법이 변경되며, 기본 Routine 은 Alloc 입니다.

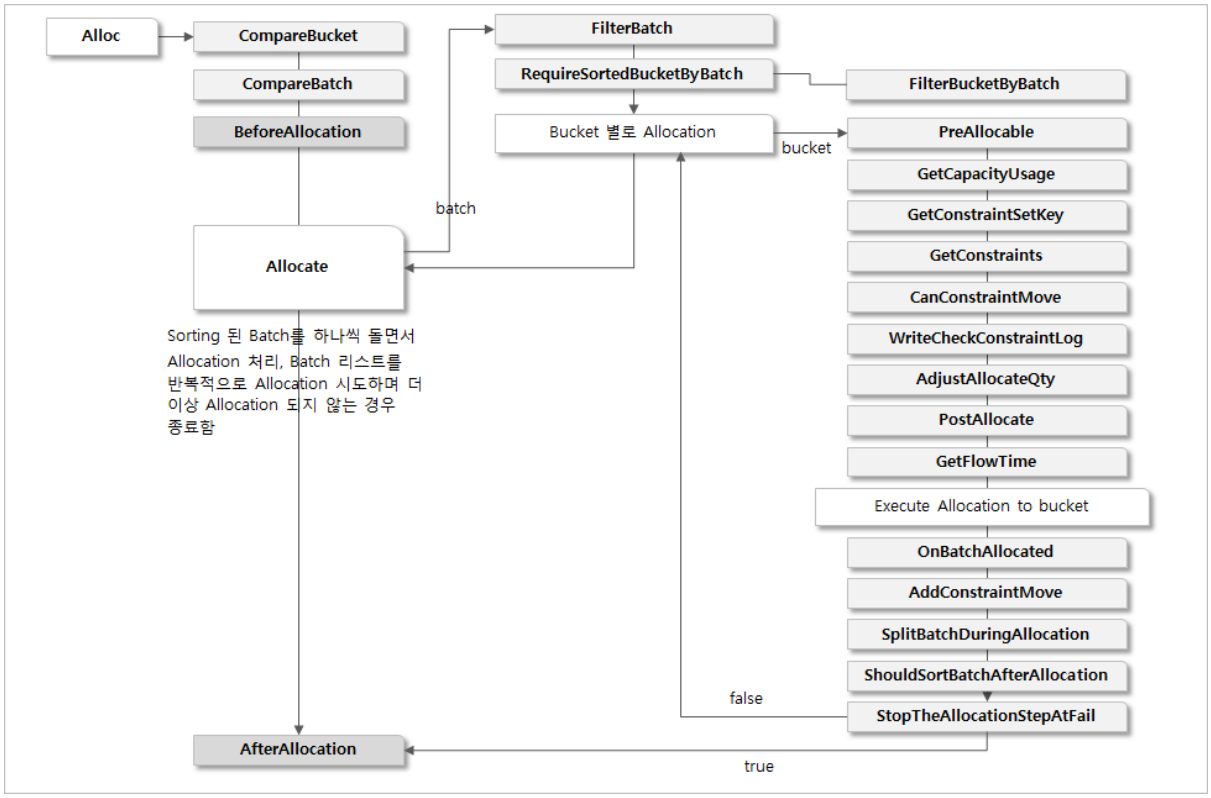
Alloc

작업물의 우선순위를 먼저 평가한 후 Bucket 순서에 따라 가용한 Bucket 에 Allocate 하는 방식입니다. 작업물이 Resource 를 선택하는 방식(Push)으로 Plan을 생성합니다.



AllocBfs

Bucket 을 먼저 선택한 후 해당 Bucket 에 로딩이 가능한 Batch 중에 우선순위와 제약을 고려하여 순차적으로 할당하는 방식입니다. Resource(장비)입장에서 작업물을 선택하는(Pull) 방식으로 Plan 을 생성합니다.



- **CompareBucket** : Allocation 대상 Bucket 의 우선순위를 결정하기 위해 Comparer 를 구현합니다.

Default Code Sample

```
public int COMPARE_BUCKET_DEF(CbsCompareInfo info, CbsBucket x, CbsBucket y, ref bool ha
ndled, int prevReturnValue)
{
    if (object.ReferenceEquals(x, y))
        return 0;

    return CompareBucketDef(x, y);
}
```

- **CompareBatch** : Allocation 대상 Batch(재공)의 Allocation 우선순위를 결정하기 위해 Comparer 를 구현합니다.

Default Code Sample

```
public int COMPARE_BATCH_DEF(CbsCompareInfo info, CbsBatch x, CbsBatch y, ref bool handl
ed, int prevReturnValue)
{
    if (object.ReferenceEquals(x, y))
        return 0;
    if (info!= null && info.WeightEval != null)
```

```
        return CompareByWeightSum(x, y);
    return CompareBatchDef(x, y);
}
```

- **BerforeAllocation** : Allocation 직전에 호출되는 FEAction 입니다.
- **FilterBatch** : 평가대상 Batch 를 Allocation 대상에서 제외할 지 평가합니다. false 인경우 해당 배치는 Allocation 되지 않습니다. Bucket 에 상관없이 Filtering 됩니다.
- **RequireSortedBucketByBatch** : 기본적인 Bucket 간의 우선순위를 사용하지 않고, Batch 별로 할당할 Bucket 의 우선순위를 결정할 필요가 있는 경우 true 를 반환합니다. true 를 반환하는 경우 등록된 CompareBucket Action 을 통해 Bucket 리스트가 재정렬 됩니다.
- **FilterBucketByBatch** : 특정 Batch에서 사용할 수 있는 Bucket 을 다시 반환합니다. 별도의 구현이 없는 경우 Allocation 가능한 모든 Bucket 을 반환합니다.
- **PreAllocable** : 특정 Bucket 에서 대상 Batch 를 Allocation 할수 있는지 여부를 판별합니다. Resource 의 시간별 제약이나 Second Resource 와 같은 제약에 의해 Allocation 이 가능한지 여부를 판별하도록 로직을 구성할 수 있습니다.

Default Code Sample

```
public bool PRE_ALLOCABLE_DEF(CbsAllocator allocator, CbsBucket bucket, CbsBatch batch,
 DateTime nowDt, ref bool handled, bool prevReturnValue)
{
    return bucket.CurrentTime >= batch.LastStepTime;
}
```

- **NeedsTobeSetup** : Setup 필요여부를 판별하기 위해 사용되는 FEAction입니다.
- **GetCapacityUsage** : Batch를 구성하는 작업물 한 Unit 이 점유하는 Capacity 의 량을 지정합니다.

Default Code Sample

```
public double GET_CAPACITY_USAGE_DEF(CbsBatch batch, CbsBucket bucket, ref bool handled,
double prevReturnValue)
{
    return batch.CurrentStep.DefaultCapacityUsage;
}
```

- **GetConstraintSetKey** : Batch 에 적용되는 제약집합의 Key 를 반환하도록 구현합니다. 이미 등록된 Set 의 경우 Constraints 를 매번 검색하지 않고 Cache 에 저장해두고 사용하여 속도를 개선할 수 있습니다.

- **GetConstraints** : Batch 에 적용되는 제약집합을 반환합니다.
- **CanConstraintMove** : 평가 대상 Batch 가 개별 제약을 만족하는지 여부를 판별합니다. 기본적으로 Constraint 의 유형에 기본 로직이 구현되어 있으나 추가적인 로직이나 사용자가 정의한 Constraint 에 대해 이를 구현할 수 있습니다.

Default Code Sample

```
public bool CAN_CONSTRAINT_MOVE_DEF(CbsCapacityConstraint constraint, CbsBatch hb, ref d
ouble qty, DateTime now, CbsBucket bucket, ref DateTime? releaseTime, ref bool handled,
 bool prevReturnValue)
{
    if (constraint is CbsDummyConstraint)
        return true;
    return constraint.CanMove(now, ref qty);
}
```

- **WriteCheckConstraintLog** : 평가 대상 Batch 가 제약을 모두 평가한 이후에 호출됩니다. 평가결과 Allocation 실패한 경우 실패사유를 기록하거나 성공한 경우에도 로그를 남기기 위해 사용할 수 있는 FEAction 입니다.
- **AdjustAllocateQty** : 평가대상 Batch 가 Bucket Capacity 및 제약을 모두 만족하는 경우에도 실제 Allocation 가능한 수량을 결정할 필요가 있는 경우 이러한 로직을 구현하기 위한 FEAction 입니다. 예를 들어 Batch 가 500개의 Unit 작업물로 구성된 경우에 한 Batch 에서는 특정 Bucket 에 100개이상을 Allocation 할 수 없다는 제약과 같이 동일 Batch 내에서 Allocation 수량을 결정하는 Rule 이 있는 경우 이러한 Rule 을 구현할 수 있습니다.
- **PostAllocate** : Batch 가 Allocation 될 것으로 결정된 직후에 호출되는 FEAction 입니다.
- **NeedsTobeSetup** : Batch 가 할당될 때 Setup 이 필요한지 여부를 반환하며, Setup 이 필요한 경우 Setup 시간을 고려해서 Setup 완료시간을 설정해야 합니다.
- **GetFlowTime** : Batch 가 실제 Bucket 에 Allocation 될때 해당 Bucket 에서 Processing 되는 시간 정보를 반환합니다. 구현을 하지 않으면 Bucket Cycle Time / Bucket Capacity 만큼의 시간이 자동 반환됩니다. 한 제품이 여러개의 Step 을 통해 작업이 완료되는 경우에 공정별 계획을 수립할때 공정의 선후조건과 계획시간을 업데이트하기 위해 필요합니다.
- **GetNextBucketAllocableTime** : 특정 Batch가 할당되는 시점에 할당된 Bucket의 다음 할당 가능한 시점을 설정합니다. 해당 FEAction에서 반환된 시간으로 Bucket의 CurrentTime 정보가 재설정됩니다.

Default Code Sample

```
public DateTime GET_NEXT_BUCKET_ALLOCABLE_TIME_DEF(CbsBucket bucket, CbsBatch batch, Dat
eTime startTime, double qty, double usage, ref bool handled, DateTime prevReturnValue)
{
    return startTime.AddSeconds(bucket.TimeUnit * qty * usage);
}
```

- **GetNextBatchAllocableTime** : 특정 Batch가 할당되는 시점에 할당 대상 Batch가 다음 공정 할당 가능한 시각을 설정하기 위해 사용되는 FEAction입니다. 해당 FEAction에서 반환된 시간은 Batch의 LastStepTime을 재설정합니다.

Default Code Sample

```
public DateTime GET_NEXT_BATCH_ALLOCABLE_TIME_DEF(CbsBucket bucket, CbsBatch batch, Date
Time startTime, DateTime nextBucketAllocableTime, double qty, double usage, Time flowTim
e, ref bool handled, DateTime prevReturnValue)
{
    return nextBucketAllocableTime.AddSeconds(flowTime.TotalSeconds);
}
```

- **OnBatchAllocated** : 대상 Batch 가 특정 Bucket 에 할당되고 Batch 의 PlanInfo 와 Bucket 의 상태가 모두 업데이트된 시점에 호출됩니다. 일반적으로 이 시점에 Bucket 의 Allocation 이력을 기록하는 로직을 개발합니다.
- **AddConstraintMove** : Batch 가 할당된 경우 해당 Batch 의 소요량만큼 모든 제약의 설정된 Capacity 를 차감해야 합니다. 기본로직에서는 할당된 Unit 수량 * CapacityUsage 에서 설정된 양의 Capacity 를 차감하도록 구현되어있습니다. Constraint 별 속성을 다르게 업데이트할 필요가 있을 경우 사용자 로직을 구현합니다.

Default Code Sample

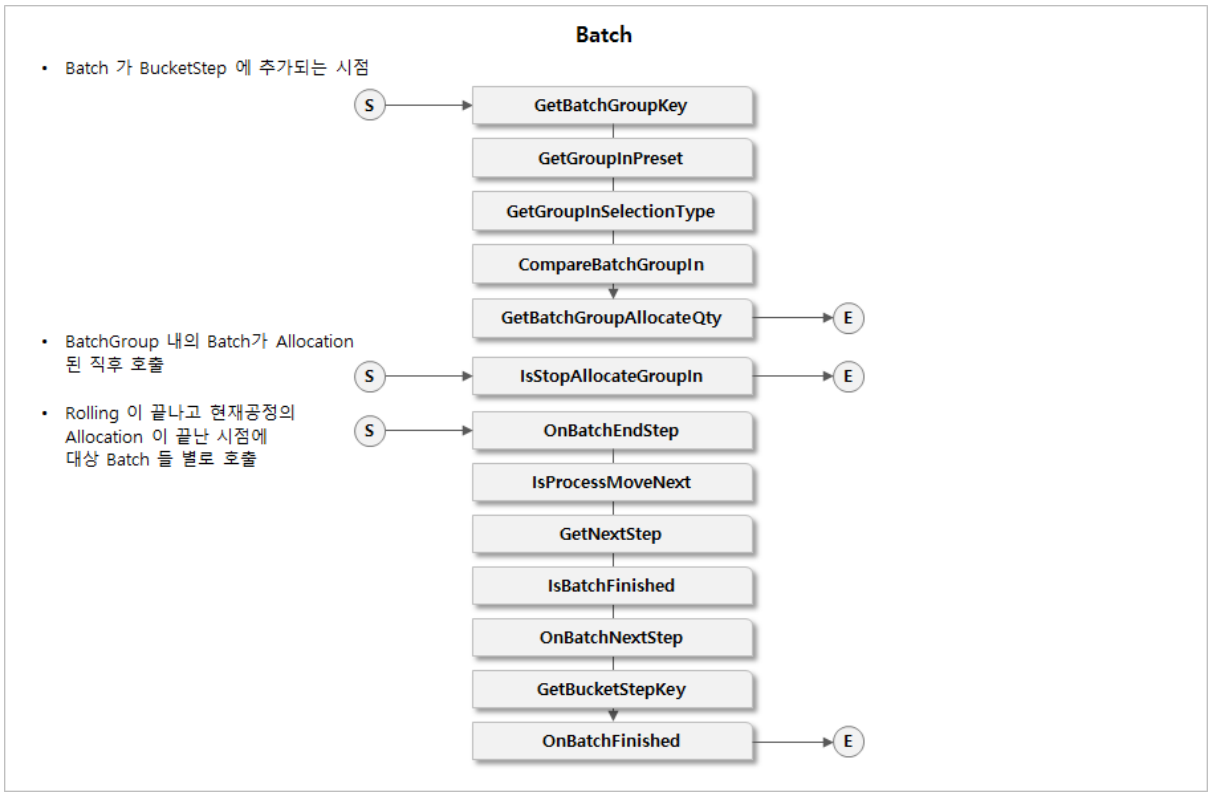
```
public void ADD_CONSTRAINT_MOVE_DEF(CbsCapacityConstraint constraint, CbsBatch hb, doubl
e qty, CbsBucket bucket, ref bool handled)
{
    if (constraint is CbsDummyConstraint)
        return;
    constraint.AddMove(qty);
}
```

- **SplitBatchDuringAllocation** : Allocation 이 끝난 이후 잔여량이나 특정 Rule 에 따라 Batch 를 Split 해야 하는 경우 Split 된 Batch 를 생성하여 반환합니다.
- **ShouldSortBatchAfterAllocation** : 만일 한 Batch 를 할당 한 후 각 Batch, Plan, Bucket 의 상태에 따라 할당 대상이 되는 Batch 들의 우선순위 재평가가 필요한 경우에는 본 FEAction 에 true 를 반환합니다. 이경우에 대상 Batch List 는 등록된 CompareBatchMethod 나 Preset 에 따라 다시 소팅됩니다.

- **StopTheAllocationStepAtFail** : 특정 Batch 가 Allocation 에 실패한 경우 순서상 이후 Batch 에 대해 더이상 Allocation 을 진행하지 않고 종료할지 여부를 판단합니다. true 를 반환하는 경우 Allocator 의 Batch 중 해당 Batch 이후 우선순위를 가지는 Batch는 Allocation 하지 않습니다.
- **AfterAllocation** : 모든 Batch 의 Allocation 이 끝난 후 호출됩니다. Bucket 에 더이상 Allocation 된 Batch 가 없을때 Allocation 이 종료됩니다.

Cbs Batch 컨트롤

CbsBatch 및 BatchGroup을 제어하기 위한 FEComponent 입니다.



1. **GetBatchGroupKey** : 동일 작업조건(동일제품)의 Batch 가 수량이 많아 비교대상 배치 수량 증가로 속도에 영향을 미치는 경우 동일 작업조건의 작업물을 하나의 BatchGroup 으로 묶어 BatchGroup 단위의 평가를 통해 수행속도를 개선할 수 있습니다. 본 Action 은 Batch 가 BucketStep 에 추가되는 시점에 함께 묶일 BatchGroup 의 Key 를 반환하는 함수입니다. 결과값을 반환하면 동일한 Key 값을 가지는 Batch 들을 해당 Key 값을 가지는 BatchGroup 으로 묶어서 하나의 Batch 처럼 처리합니다. Key 값을 반환하지 않으면 BatchGroup 으로 묶지 않습니다.

2. **GetGroupInPreset** : BatchGroup 이 생성되는 시점에 해당 BatchGroup 내에서 사용될 Batch 간 평가를 위한 Preset ID 를 반환합니다. 결과값을 반환하지 않는 경우에는 Group 내의 Batch 들은 CompareBatchGroupIn 에서 구현된 함수에 따라 소팅합니다. 실제 사용을 위해서는 Preset 이 사전에 등록되어 있어야 함(Preset 은 Init/GetWeightPresets Action 을 통해 등록합니다.)

3. **GetGropuInSelectionType** : BatchGroup 내의 Batch 들 간의 우선순위를 어떻게 평가할 것인지에 대한 타입 값입니다. 반환 타입은 Mozart.Seeplan.Cbsim.SelectionType = {Custom, WeightSorted, WeightSum} 값입니다. Custom 을 선택하는 경우 CompareBatchGroupIn 함수를 사용합니다. WeightSorted, WeightSum 을 선택하는 경우 GetGroupInPreset 에서 사용할 Preset ID 를 반환해야 합니다. WeightSorted 는 Preset 을 사용하여 순차 소팅 방식으로 Batch 를 평가하여 소팅합니다. WeightSum 을 선택하는 경우 설정된 Preset을 사용하여 가중합 방식으로 정렬합니다.

4. **CompareBatchGroupIn** : Group의 SelectionType = Custom 인 경우에 사용되는 소팅 로직을 구현합니다.

   Default Code Sample

```
public int COMPARE_BATCH_GROUPIN_DEF(CbsBatchGroup bgroup, ICbsBatch x, ICbsBatch y,
ref bool handled, int prevReturnValue)
{
    if (object.ReferenceEquals(x, y))
        return 0;

    int cmp = x.Sample.LastStepTime.CompareTo(y.Sample.LastStepTime);
    if (cmp == 0)
        cmp = x.Sample.LotID.CompareTo(y.Sample.LotID);

    return cmp;
}
```

5. **GetBatchGroupAllocateQty** : BatchGroup 내에서 특정 Rolling 시점에 Allocation 가능한 수량을 지정합니다. Batch가 추가될 때마다 호출되어 수량을 다시 지정할 수 있습니다. 해당 그룹에서 1회 Rolling 시점에 Allocation 가능한 최대 수량입니다. 만일 IsStopAllocateGroupIn 함수에서 true 값을 반환하지 않고 Bucket 의 Capacity 가 충분한 경우 해당 그룹에서는 설정된 수량만큼 Allocation 됩니다.

6. **IsStopAllocateGropuIn** : 동일 BatchGroup 내의 우선순위 Batch 가 Allocation 된 직후 호출됩니다. true 값을 반환하면 GetBatchGroupAllocateQty 에서 설정된 수량 이하로 할당된 경우라도 해당 Group 에서 더 이상 Allocation 할 Batch 를 선택하지 않고 해당 BatchGroup 의 할당을 종료합니다.

7. **OnBatchEndStep** : Batch 가 특정 Step 을 완료하는 시점에 호출되는 Event 함수입니다.

8. **IsProcessMoveNext** : Batch 가 특정 Step 을 끝낸 후 다음 Step 으로 진행할지 여부를 반환합니다. 기본 값은 true 입니다. 만일 false를 반환하는 경우 Batch 는 다음 공정으로 이동하지 않고 동일한 Step 을 반복적으로 Allocation 하게 됩니다. Flow 상의 여러 공정이 실제 해당 공정들을 처리하는 장비그룹(BucketStep)이 같은 경우 공정처리를 반복적으로 할 수 있습니다.

9. **GetNextStep** : Batch 의 현재 Step 이후 처리해야 할 공정을 반환합니다. 이 함수를 통해 제품별로 Skip 로직을 처리할 수 있습니다.

   Default Code Sample

```
public CbsStep GET_NEXT_STEP_DEF(ICbsBatch batch, CbsLoadInfo loadInfo, CbsStep ste
p, ref bool handled, CbsStep prevReturnValue)
{
    var next = (CbsStep)step.GetDefaultNextStep();
    return next;
}
```

10. **IsBatchFinished** : Batch 의 현재 공정이 완료공정인지 여부를 반환합니다. 현공정의 완료 시점에 판단하여 Batch 를 최종 완료 처리합니다.

11. **GetBucketStepKey** : 제품의 생산 Flow 상의 공정과 공정을 처리하는 BucketStep(BucketGroup)의 명칭이 다른 경우 특정 Step 을 처리할 자원그룹(BucketStep)의 명칭(Key)을 반환하여 해당 공정으로 이동할 수 있도록 처리합니다.

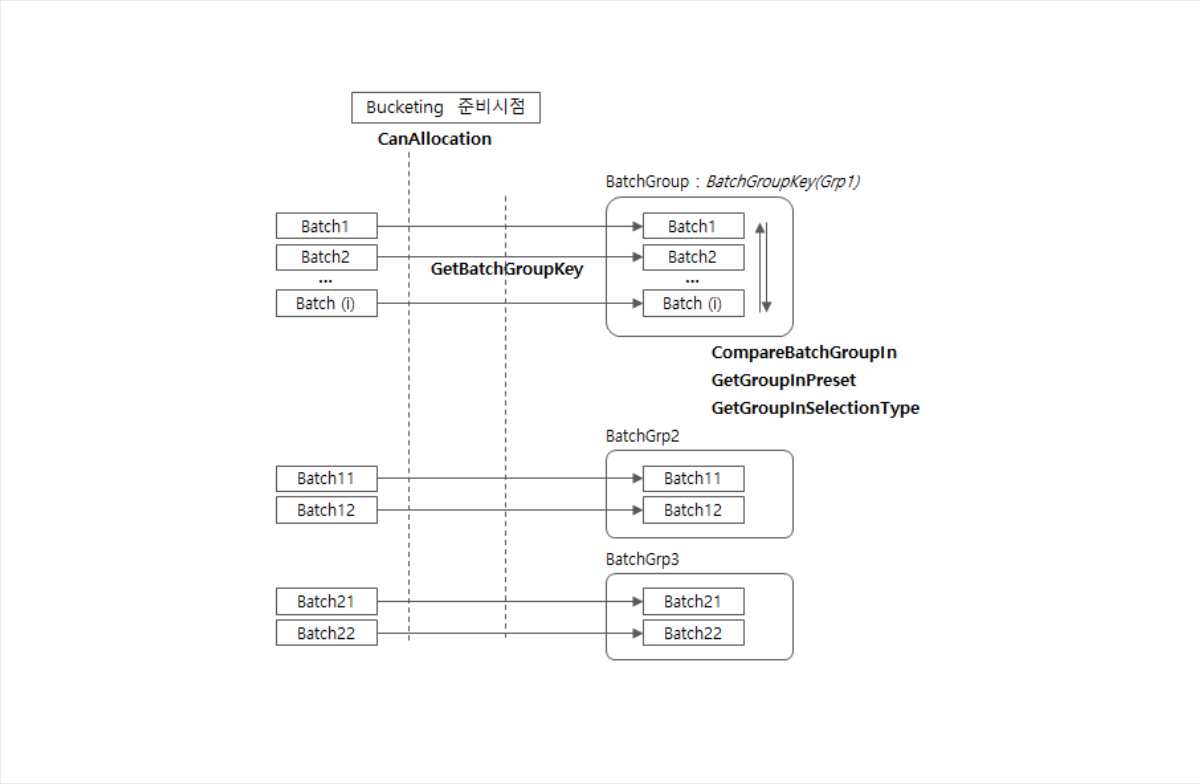
    Default Code Sample

```
public string GET_BUCKET_STEP_KEY_DEF(ICbsBatch batch, ref bool handled, string prev
ReturnValue)
{
      if (batch != null)
         return batch.CurrentStepID;
      return default(string);
}
```

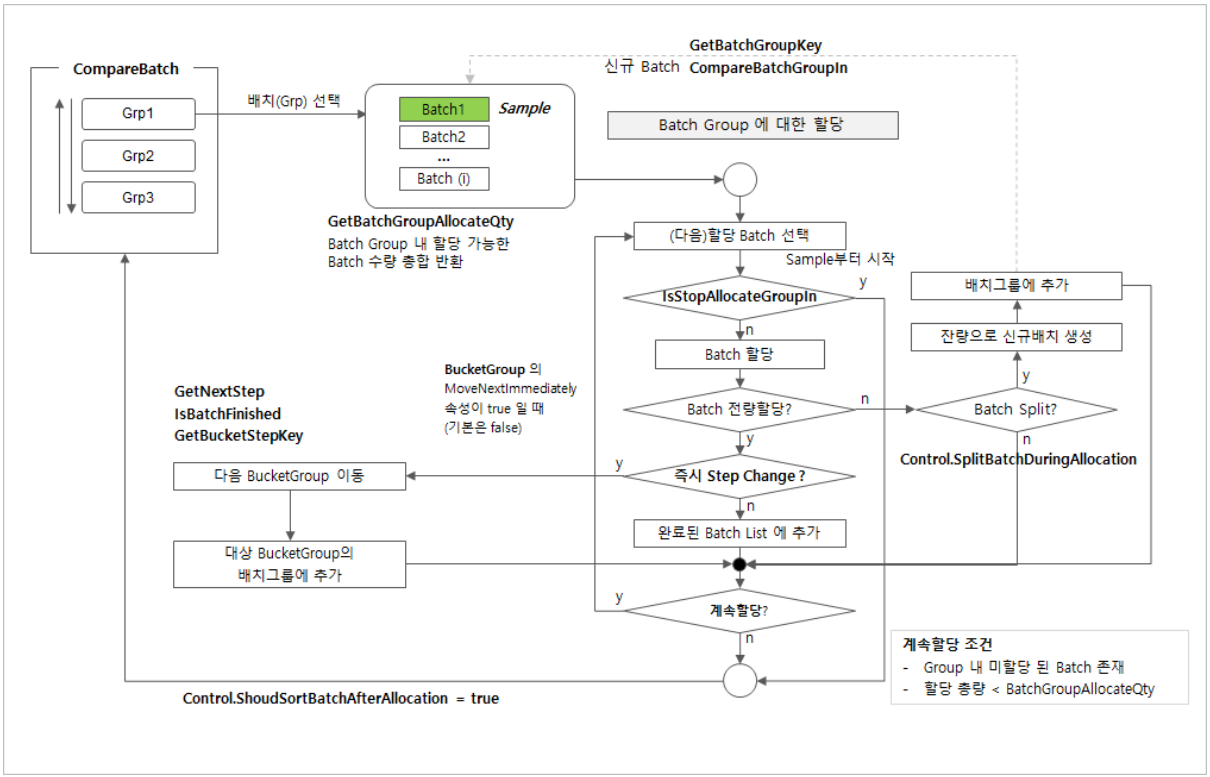
12. **OnBatchNextStep** : GetNextStep 에서 반환된 Step 정보로 이동한 직후 호출되는 Event 함수입니다.

13. **OnBatchFinished** : Batch 가 모든 공정을 완료 처리한 시점에 호출되는 Event 함수입니다.

«Batch와 BatchGroup의 관계»

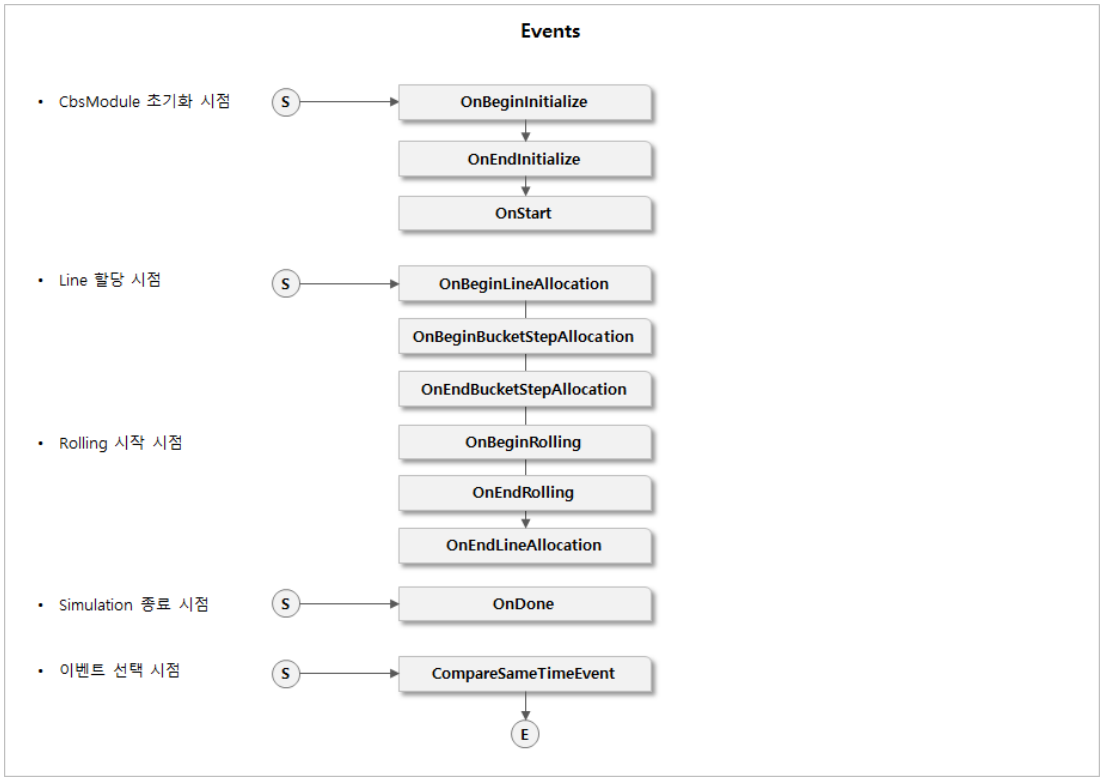


«BatchGroup의 할당 절차»



Cbs Events 컨트롤

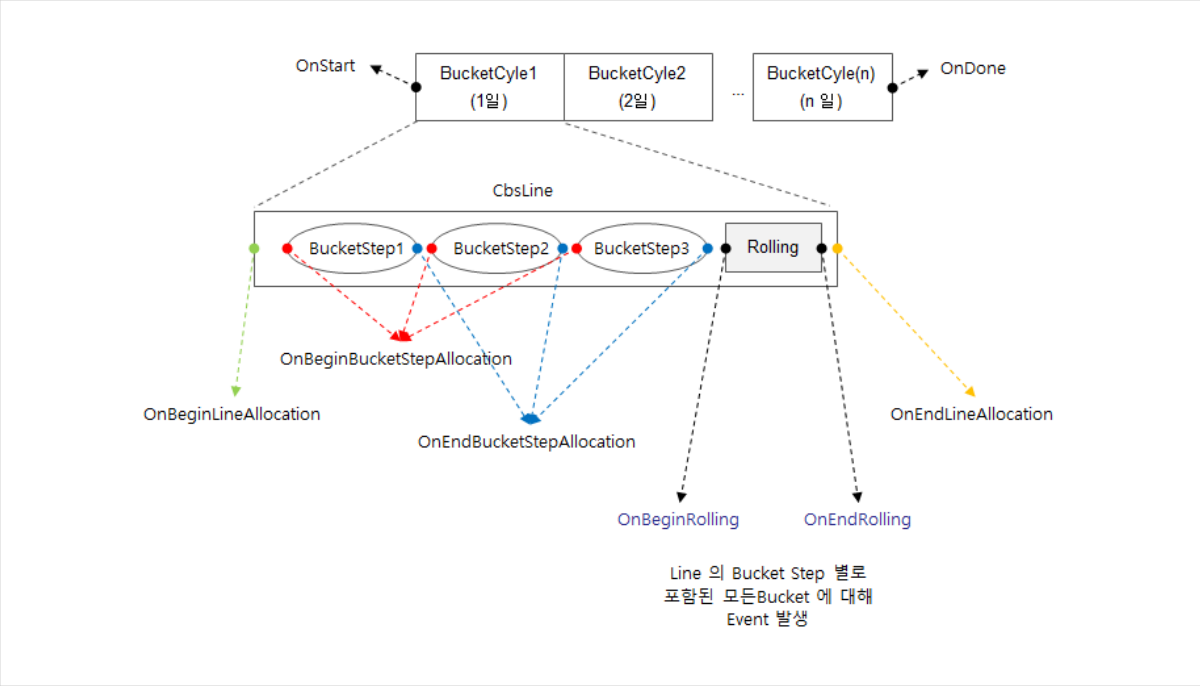
Cbs Module 실행시 발생하는 Global 한 Event 시점(Day Change, Shift Change, ..)에 사용자 정의 코드를 작성할 수 있도록 FEAction 을 제공합니다.



- **OnBegineInitialize** : Cbs Module 의 초기화 로직 시작전에 호출됩니다.
- **OnEndInitialize** : Cbs Module 의 초기화 로직이 모두 처리된 직후에 호출됩니다.
- **OnStart** : Cbs 엔진이 시작하는 시점에 호출됩니다. 초기화는 모두 끝난 상태입니다.
- **OnBeginLineAllocation** : Line 별 Allocation 시작시점에 호출되는 Event 함수입니다. 대부분의 경우 CbsLine을 1개로 설정하게 됨으로 전체 Allocation이 시작되는 시점에 한 번 호출되는 것이 일반적입니다. 만일 복수개의 CbsLine으로 구성을 하는 경우 Line 별 Allocation 시작시점에 호출됩니다. Allocation은 Rolling 횟수만큼 수행됩니다. 즉 일단위 Cycle로 7일을 수행하면 Line 단위 Allocation이 총 7번 실행됨으로 본 함수는 매일 Allocation 시작되는 시점에 7번 호출되게 됩니다.
- **OnBeginBucketStepAllocation** : BucketStep의 매 할당 시작전에 호출되는 Event 함수입니다.
- **OnEndBucketStepAllocation** : BucketStep의 매 할당 종료시점에 호출되는 Event 함수입니다.

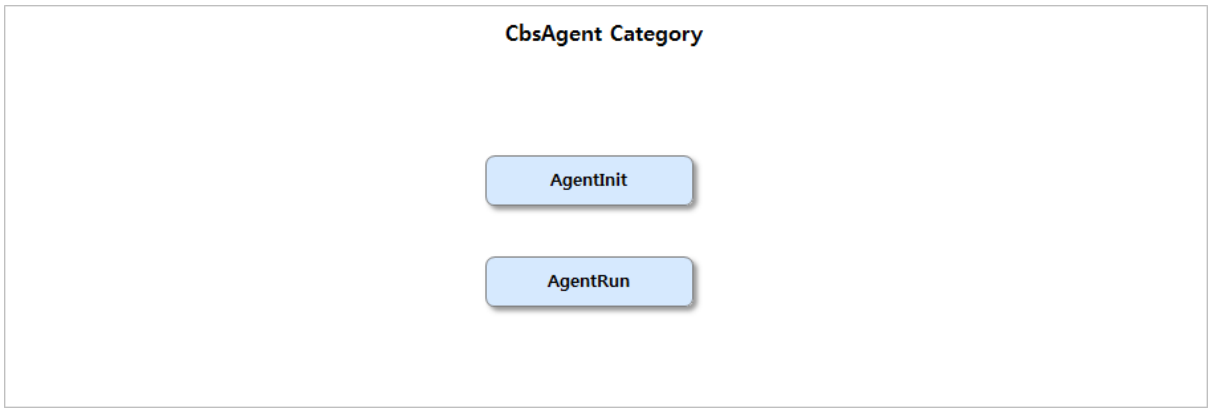
- **OnEndLineAllocation** : Line 별 Allocation 종료시점에 호출되는 Event 함수입니다.
- **OnBeginRolling** : Rolling 시작시점에 호출되는 Event 함수입니다. Rolling의 의미는 한 Bucket 구간의 할당이 모두 끝난 후 Bucket과 Constraint의 Capacity를 Reset 하는 작업을 의미합니다.
- **OnEndRolling** : Rolling 종료시점에 호출되는 Event 함수입니다.
- **OnDone** : Cbs 엔진이 수행을 모두 마치고 종료하는 시점에 호출됩니다.
- **CompareSameTimeEvent** : 같은 시간에 발생하는 이벤트들이 있을 때 우선순위를 설정하기 위한 FEAction입니다. 매 이벤트가 선택되는 시점마다 호출이됩니다.

«이벤트 순서»



CBS Agent

CbsAgent 의 제어를 위한 FEComponent 그룹입니다.



AgentInit

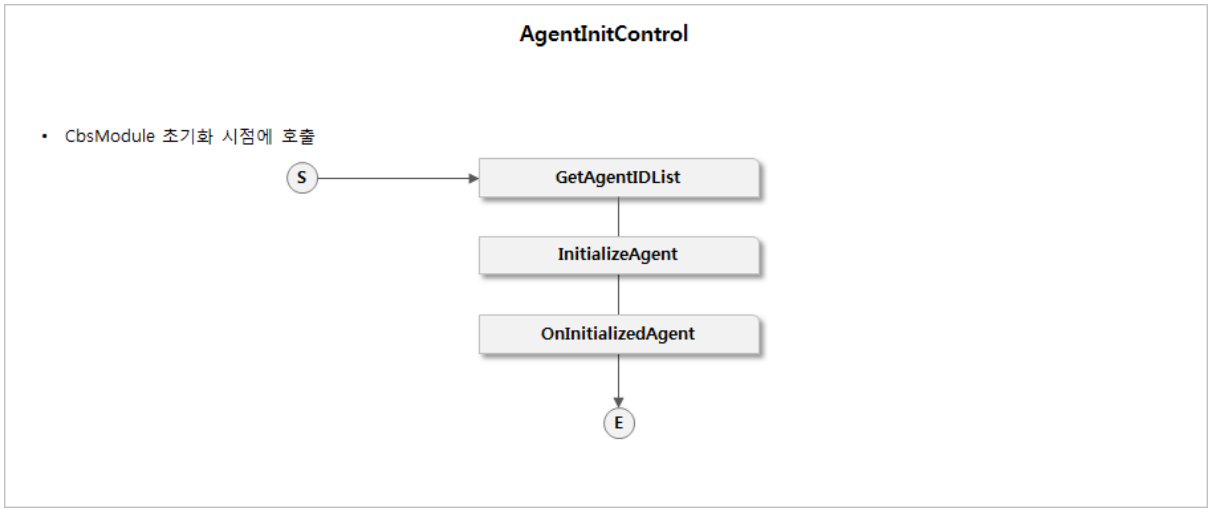
CbsModule 에서 사용할 Agent 를 생성하고 초기화하기 위한 FEComponent 입니다.

AgentRun

Agent 의 실행을 제어하기 위한 FEComponent 입니다.

Cbs Agent Init 컨트롤

CbsModule 에서 사용할 Agent 를 생성하고 초기화하기 위한 FEComponent 입니다.

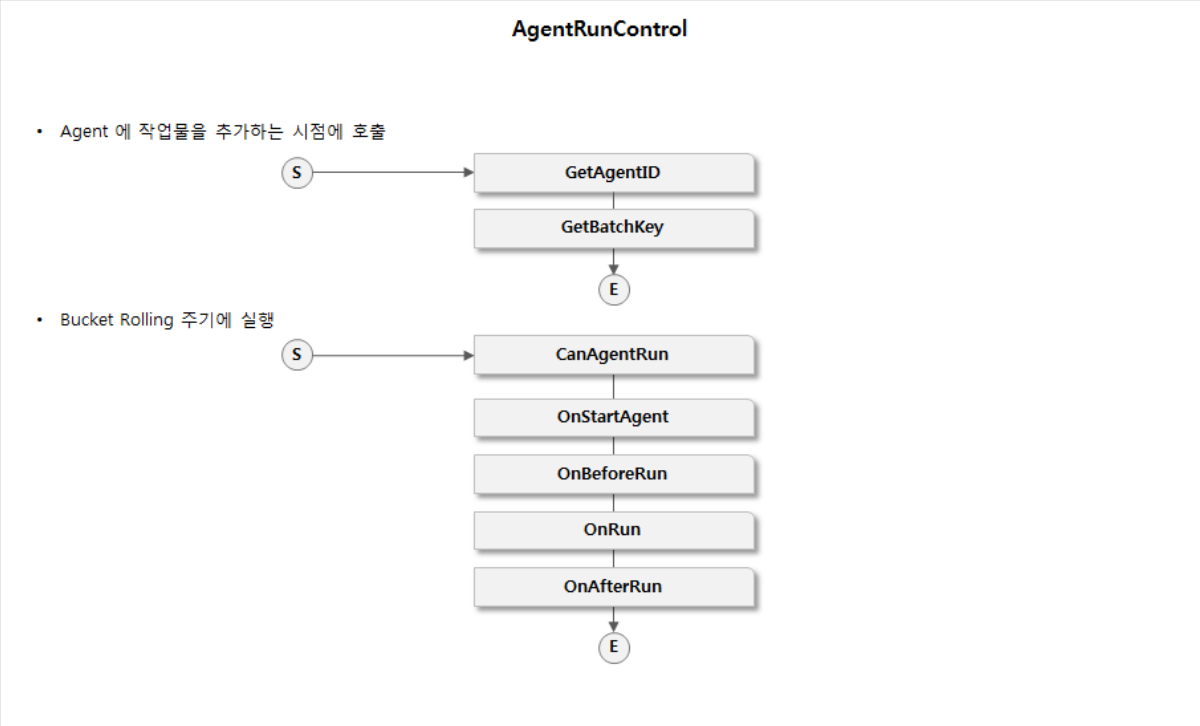


1. **GetAgentIDList** : Cbs 모듈에서 사용할 Agent 의 ID 리스트를 반환합니다. 리스트의 항목별로 Agent 를 생성합니다. Agent 의 생성부터 실행에 대한 제어는 CbsAgentControl 을 통해 수행가능합니다.

2. **InitializeAgent** : Agent 생성 직후 호출됩니다. Agent 에 설정할 기본적인 속성을 정의할 수 있습니다.
3. **OnInitialzedAgent** : Agent 생성 이후 모듈에서 Allocation 대상이 되는 작업물(Batch)를 초기화 한 이후에 호출됩니다. 이때에는 Agent 에 초기 Batch 가 추가된 상태입니다.

Cbs Agent Run 컨트롤

Cbs 모듈을 실행시 자원그룹인 BucketStep 의 전, 후에 공정(CbsStep, BucketStep)과 독립적으로 재공(Batch)를 적재하고 Release 할 수 있도록 구성된 Agent 로직을 제어하기 위한 Component 입니다.



1. **GetAgentID** : CbsAgentManager 에 배치를 등록하는 시점에 배치를 저장하기 위한 Agent ID 반환하는 FEAction 입니다. 배치를 Agent 에 추가하는 것은 사용자가 CbsModule 의 초기화 시점에 직접 처리해야 합니다. 일반적으로 Batch를 초기화하는 시점에 특별한 작업을 하지 않는 경우 모든 Batch는 초기 Step 을 처리하기 위한 BucketStep 에 등록이 되지만, Agent 를 사용해서 투입의사결정이나 공정간의 재고를 모델링하기 위해서는 Agent 에 추가하는 방식을 사용합니다.
2. **GetBatchKey** : Agent 에 등록되는 Batch 가 등록되는 시점에 batch 를 그룹핑하여 저장하기 위한 Key 입니다. Agent 에 Key 별로 Stock 을 구성해서 batch 를 저장하고 있습

니다.

3. **CanAgentRun** : Rolling 시점에 등록된 Agent 별로 실행 여부를 반환합니다. false 를 반환하는 경우 해당 Rolling 구간에서 지정된 Agent 로직은 수행하지 않습니다.
4. **OnStartAgent** : Agent 가 최초 Run 을 수행하는 시점에 한번 호출 됩니다.
5. **OnBeforeRun** : Agent 수행시점(Rolling)마다 호출되며, Agent의 주요 로직 실행 전에 호출됩니다.
6. **OnRun** : Agent 수행 Main 로직을 구현하기 위한 FEAction 입니다. 본 Action 에 구현부가 없는 경우 Agent 는 아무일도 수행하지 않습니다.
7. **OnAfterRun** : Agent 로직 수행 직후에 호출됩니다.