

# 프로젝트명 : EQ-1 Core

## 범용 비전 검사 소프트웨어 프레임워크

### 프로젝트 목적

eq1\_core는 다양한 도메인의 비전 검사기를 빠르게 개발할 수 있도록 설계된 **범용 Core 프레임워크**입니다.

검사 방식(단일샷, 멀티샷, 연속 검사)과 각 컴포넌트(카메라, 네트워크, 엔진 등)를 조합하여 유연한 애플리케이션을 구성할 수 있습니다.

### 주요 특징

- **Core**는 EQ1의 핵심로직인 연속된 프레임의 검사 로직을 패키지로 제공
- **Core**가 모든 컴포넌트의 DI/라이프사이클 관리 담당
- **Core**는 내부적으로 Camera, Engine을 사용하여 검사 파이프라인을 구성
- **DB** 등 외부 설정을 기반으로 검사 파이프라인을 동적으로 구성
- **Application** 계층을 통해 도메인에 맞는 유저 정의 가능
- **Application** 계층을 통해 이벤트 기반 Network 처리 가능
- 플러그인 방식의 엔진 등록 시스템으로 확장성과 재사용성 강화
- 의존성 역전을 통한 느슨한 결합과 테스트 용이성
- 추후 CLI 및 GUI 모드 확장 고려

### 프로젝트 구조

```
eq1_core/
├── src/
│   ├── core.py                # 메인 Core 클래스
│   ├── engines/
│   │   ├── interface.py      # BaseEngine 인터페이스
│   │   ├── factory.py        # 엔진 팩토리 (의존성 역전)
│   │   └── sample.py         # 샘플 엔진 구현
│   ├── domain/
│   │   ├── entities/        # 도메인 엔티티 및 포트
│   │   │   └── entities.py   # 도메인 엔티티
│   │   ├── ports/
│   │   └── services/         # 데이터 접근 로직 분리
│   ├── infrastructure/       # 외부 데이터 시스템 연동
│   │   ├── db/              # DB 관련 코드
│   │   └── api/
│   ├── inspection/
│   │   ├── event_listener.py
│   │   └── worker.py
│   ├── frame_number.py        # 프레임 번호 관리
│   ├── data.py               # 데이터 클래스
│   ├── plugin.py             # 플러그인 시스템
│   └── decorators.py          # 데코레이터 (예: CLI 명령
```

```

├── events.py           # 이벤트 시스템
├── logger.py          # 로깅 시스템
├── example/
│   ├── multishot_mode/
│   │   └── main.py     # 다중 촬영 모드 애플리케이션 예시
│   └── continuous_mode/
│       └── main.py     # 연속 모드 애플리케이션 예시
├── lib/               # 외부 라이브러리 (카메라 등)
└── README.md

```

## 설치 방법

### 요구사항

- Python 3.8+
- MySQL 데이터베이스
- 필요한 의존성 패키지

### 환경 설정

```

# 환경 변수 설정
export
DB_URL="mysql+pymysql://username:password@localhost:3307/database"
export LOG_PATH="./public/logs"

```

### 데이터베이스 설정

- MySQL 데이터베이스가 설치되어 있어야 합니다
- 필요한 테이블 스키마가 생성되어 있어야 합니다
- 카메라, 엔진, 컴포넌트 등의 설정 데이터가 있어야 합니다

## 사용 예시

### 기본 연속 모드 애플리케이션

```

import os
from src.core import Core
from src.data import InspectionMode
from src.infrastructure.db.repositories import DBRepositorySet
from src.engines.sample import SampleRollSurfaceEngine

# 환경 설정
os.environ["DB_URL"] = "mysql+pymysql://user:pass@localhost:3307/eq1"
os.environ["LOG_PATH"] = "./logs"

```

```

class MyInspectionApp:
    def __init__(self):
        # Core 인스턴스 생성
        self._core = Core(
            stage_name="STAGE_1",
            product_code="SAMPLE-PRODUCT",
            repos=DBRepositorySet,
            mode=InspectionMode.CONTINUOUS
        )

        # 엔진 등록
        self._core.register_engine("sampleengine",
SampleRollSurfaceEngine)

        # 이벤트 핸들러 등록
        self._setup_event_handlers()

    def _setup_event_handlers(self):
        from src.events import InspectionEvents

        self._core.register_hook(
            event_name=InspectionEvents.ComponentFinished().name,
            hook=self.on_component_finished
        )

    def on_component_finished(self, **kwargs):
        """컴포넌트 검사 완료 시 호출되는 핸들러"""
        inspection_result = kwargs.get("inspection_result_data")
        component_result = kwargs.get("component_result_data")

        print(f"검사 완료: {component_result.component_name}")
        print(f"결과: {inspection_result.result}")

    def run(self):
        """애플리케이션 실행"""
        self._core.start()

# 애플리케이션 실행
if __name__ == "__main__":
    app = MyInspectionApp()
    app.run()

```

## 커스텀 엔진 개발

```

import numpy as np
from typing import Tuple
from src.engines.interface import BaseEngine
from src.data import RollSurfaceEngineResult, RollSurfaceEngineParams

class CustomEngine(BaseEngine):

```

```

    @classmethod
    @property
    def name(cls) -> str:
        return "CustomEngine"

    def get_results(self, image: np.ndarray) -> Tuple[bool,
RollSurfaceEngineResult]:
        # 커스텀 검사 로직 구현
        # AI 모델 추론, 이미지 처리 등

        result = RollSurfaceEngineResult(
            is_ok=True,
            is_failed=False,
            base_engine_name=self.name,
            engine_params=RollSurfaceEngineParams(),
            # ... 기타 결과 데이터
        )

        return False, result # is_failed, result

    @classmethod
    def create_from_config(cls, config: dict):
        """설정 기반 인스턴스 생성"""
        return cls()

# 엔진 등록
core.register_engine("custom", CustomEngine)

```

## CLI 명령어 등록

```

from src.decorators import cli_command

@cli_command("shot")
def trigger_inspection(core: Core) -> bool:
    """수동 검사 트리거"""
    # 특정 카메라에 소프트웨어 트리거 전송
    return core.trigger_camera("CAMERA_SERIAL")

@cli_command("next")
def next_lot(core: Core) -> bool:
    """다음 LOT으로 변경"""
    return core.emit("lot_change_requested")

```

## 확장 방법

### 1. 새로운 엔진 개발

```
# engines/my_engine/engine.py
from src.engines.interface import BaseEngine

class MyCustomEngine(BaseEngine):
    # BaseEngine 인터페이스 구현
    pass

# 애플리케이션에서 등록
core.register_engine("my_engine", MyCustomEngine)
```

## 2. 이벤트 핸들러 추가

```
from src.events import CustomEvent

class MyCustomEvent(CustomEvent):
    def __init__(self):
        super().__init__(
            name="my_custom_event",
            description="나만의 커스텀 이벤트"
        )

# 이벤트 핸들러 등록
core.register_hook(
    event_name="my_custom_event",
    hook=my_event_handler
)
```

## 3. 새로운 검사 모드 추가

```
from src.data import InspectionMode

# 새로운 모드로 Core 생성
core = Core(
    stage_name="STAGE_1",
    product_code="PRODUCT_CODE",
    repos=DBRepositorySet,
    mode=InspectionMode.SINGLE_SHOT # 또는 MULTI_SHOT
)
```

## 핵심 개념

### Core 클래스

- 모든 컴포넌트의 생명주기 관리

- 이벤트 기반 통신 허브 역할
- 플러그인 방식의 엔진 관리

## 엔진 시스템

- **BaseEngine** 인터페이스 기반
- 팩토리 패턴을 통한 동적 생성
- 의존성 역전으로 느슨한 결합

## 이벤트 시스템

- 컴포넌트 간 비동기 통신
- 확장 가능한 훅 시스템
- 도메인별 커스텀 이벤트 지원

## 데이터 서비스

- Service 계층을 통한 데이터 접근 추상화
- 비즈니스 로직과 데이터 로직 분리
- 테스트 용이성 향상



## 테스트



## API 문서

X



## 기여 방법

1. 이슈 생성 또는 기존 이슈 확인
2. 기능 브랜치 생성 (**git checkout -b feature/새기능**)
3. 변경사항 커밋 (**git commit -am '새기능 추가'**)
4. 브랜치 푸시 (**git push origin feature/새기능**)
5. Pull Request 생성



## 라이선스

이 프로젝트는 [MIT 라이선스](#) 하에 배포됩니다.



## 문의사항

프로젝트 관련 문의사항이 있으시면 이슈를 생성해 주세요.