객체지향개발방법론

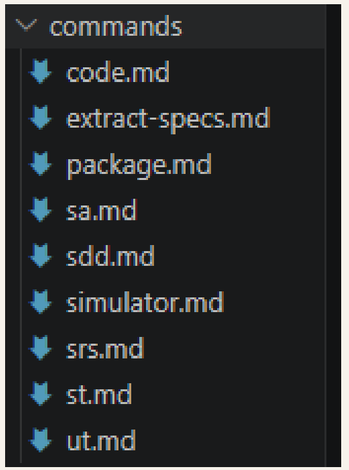
팀프로젝트#5

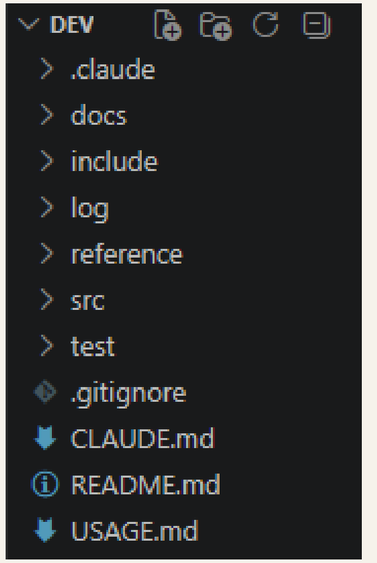
2026.05.16

7팀 강희진 김희용 양지윤 윤성진

클로드 기능 설명 및 작업환경 설정

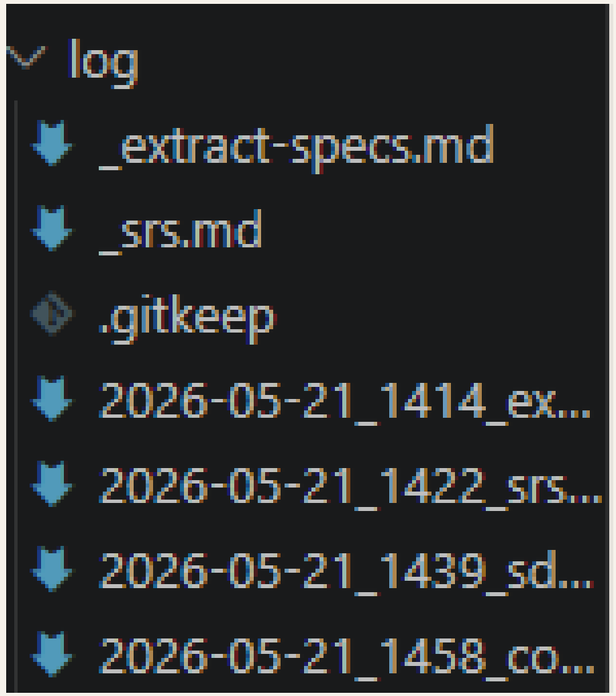
- Planning mode: 실제 코드 수정은 이루어지지 않고 수정 계획을 세우는 터미널 모드.
  - 이를 통해 작업환경을 구축하고 skill 등을 계획함.
- skill: md 파일로 정리한 작업 지시를 클로드가 실행하게 하는 명령어.
  - 각 단계의 작업을 skill로 만들고 순차적으로 실행함.
  - 사용자가 복잡한 컨텍스트를 터미널을 통해 입력하지 않아도 됨.
- 비주얼 코드 터미널에서 ‘claude’ 호출하여 진행함.
- Sonnet 모델 사용.





클로드 기능 설명 및 작업환경 설정

- Planning mode를 통해 기존 자료 바탕 작업 환경 구성
- 작업 환경에서 각 단계를 skill로 단독 분리
  - 각 skill 실행 후 log 폴터 하위에 기록 남기도록.
- “재현 가능한“ 작업 환경을 만듦.
  - 상세 작동은 USAGE.md 또는 README.md 참고



기존 자료 md화 및 재설계

- /extract-specs 호출 시 참조되는 md

```
## 1. 본 작업 – 3단계 재증류

### A. 1차 이해 (Reference 자료 통독 – 산출물 없음)

`reference/`의 모든 1차 자료를 **이해 목적으로** 읽는다. 이 단계는 산출물
파일을 만들지 않으며, Claude의 작업 컨텍스트에 도메인 모델을 형성하는 데
사용된다.

읽기 대상:
- `reference/객체지향개발방법론#1_V3.pdf`, `#2_V2.pdf`, `#3_v2.pdf`, `#4
pdf` – 팀의 명세
- `reference/objective.md` – 본 과제 목표
- `reference/CICD-Team7-main/` – 이전 학기 7팀의 코드 (디렉토리·헤더 위주,
함수 본문은 깊이 읽지 않음)

읽기 목적 (Claude가 답할 수 있어야 함):
1. 이 시스템의 **존재 이유**는 무엇인가? (한 줄)
2. 외부에서 시스템에 **어떤 자극**이 들어오는가? (액터·이벤트)
3. 시스템은 **어떤 상태들**을 가지는가? (개념적 상태, PDF 클래스명 무관)
4. **품질·환경 조건**(시간 한계, 추상화 요구, 빌드 환경 등)은 무엇인가?
5. **본질적으로 중복·과세분화된 요구사항**은 어떤 것인가? (재증류 후보)

대상 자료의 다이어그램·클래스명·메서드명·테스트 케이스명은 **읽되 기억의
닻으로만 쓴다.** 그대로 옮길 대상이 아니다.
```

```
### B. 재증류 산출 (`docs/specs/`)

읽기를 마치면 다음 3개 문서만 만든다. 원본 PDF별 1:1 변환 문서
(`01-team-project-1.md` 등)는 **만들지 않는다**.

#### B-1. `docs/specs/system-overview.md`

Claude가 작성하는 1~2페이지짜리 시스템 개요. 형식 자유, 다만 다음을 포함:

- **시스템 정의**: 한 단락(3~5줄)으로 RVC가 무엇이고 왜 필요한가를 자기
언어로 서술
- **외부 환경 모델**: 어떤 액터·센서·물리적 환경이 입력으로 들어오고 시스템이
무엇을 출력하는가 (mermaid context diagram 1개)
- **본질 동작 모드**: 시스템이 가지는 운영 모드를 Claude가 정리 (예: 정상
청소, 회피, 강화 청소, 종료 – 명칭은 자유)
- **품질 핵심**: 본 시스템 설계에서 가장 강한 제약이 되는 NFR 3~5개를 골라
강조

PDF의 Use Case Diagram, Domain Model을 **mermaid로 재현하지 않는다.**
context diagram은 외부 액터-시스템 경계 수준만.
```

```
#### B-2. `docs/specs/requirements.md` – 재증류된 FR/NFR

원본의 R1.1~R6.1, NFR-P/O/OE/I를 **Claude가 새로 정의한 ID 체계로
재진술**한다. 권장 접근:
```

```
#### B-3. `docs/specs/use-cases.md` – 재증류된 UC

원본 UC1~UC9의 Use Case Details를 **본질 흐름만 남기고 재진술**한다. 권장
접근:
```

기존 자료 md화 및 재설계

- system-overview.md, requirements.md, use-cases.md

```
# RVC 제어 소프트웨어 – 시스템 개요

## 1. 시스템 정의

RVC(Robot Vacuum Cleaner) 제어 소프트웨어는 자율 주행 로봇 청소기의 동작
전반을 관장하는 임베디드 제어 SW이다. 이 시스템은 근접 센서와 먼지 센서로부터
실시간 입력을 받아 이동 방향을 결정하고, 청소 강도를 조절하며, 이상 상황 발생
시 안전하게 종료한다. 사용자는 CLI를 통해 시스템을 기동·종료하며, 시스템은 그
사이에 완전 자율적으로 동작한다. 물리적 하드웨어에 대한 직접 의존을 추상화
계층으로 분리하여 테스트 용이성과 유지보수성을 동시에 확보하는 것이 이
소프트웨어의 핵심 설계 목표이다.
```

```
# RVC 제어 소프트웨어 – 유스케이스 명세 (재증류)

> **재구성 근거**: 원본 UC3·UC4·UC5·UC6(장애물 회피 3분기)을 단일 UC-03으로
통합하고
> 방향 결정 로직을 분기 표로 표현한다. UC8·UC9(종료)는 트리거 차이가
명확하므로 분리 유지.
> 원본 9개 UC → 재증류 6개 UC.
```

```
# RVC 제어 소프트웨어 – 요구사항 명세 (재증류)

> **작성 방침**: 본 문서의 모든 ID, 그룹, 서술은 원본 PDF의 것을 그대로
옮기지 않고 Claude가 재증류한 것이다.
> 원본 ID와의 매핑은 문서 말미의 추적성 표에 기재한다.
```

FR

1.1 시스템 생명주기 제어 (FR-CTRL)

FR-CTRL-01 – 시스템 기동
사용자가 CLI에서 전원 켜기 명령을 입력하면, 시스템은 5초 이내에 내부 구성 요소를 초기화하고 준비 완료를 표준 출력에 알린 뒤 자동 청소 루프(FR-CLEAN-01)로 진입해야 한다.

검증 기준: 전원 ON 명령 수신 후 5초 이내에 자동 청소 루프 첫 반복이 시작됨을 확인한다.

FR-CTRL-02 – 정상 종료
사용자가 CLI에서 전원 끄기 명령을 입력하면, 시스템은 진행 중인 모든 이동·청소 동작을 중지하고, 구동 모터와 청소 장치를 안전하게 정지한 뒤 종료 메시지를 출력하고 오프라인 상태로 전환해야 한다.

FR-CTRL-03 – 오류·예외 종료
시스템 내 어느 구성 요소에서든 복구 불가능한 예외 또는 오류가 감지되면, 시스템은 오류 내용을 표준 출력에 즉시 알리고 FR-CTRL-02와 동일한 정지 절차를 수행한 뒤 오프라인 상태로 전환해야 한다.

1.2 이동·방향 제어 (FR-MOVE)

FR-MOVE-01 – 전진 이동
자동 청소 루프가 활성화되면, 시스템은 구동 모터를 통해 로봇을 전방으로 이동시켜야 한다. 이동은 장애물 감지, 종료 명령, 또는 오류가 발생하기 전까지 지속된다.

FR-MOVE-02 – 장애물 회피 (3-경로 결정)
전방 장애물이 감지되면 시스템은 전진 이동을 중지하고 좌·우측 센서 상태를 조회하여 아래 결정 표에 따라 이동 방향을 전환해야 한다.

FR-MOVE-03 – 회전 중 입력 마스크
회전 동작(좌회전 또는 우회전)이 진행되는 동안에는 전방·좌·우 근접 센서 및 먼지 센서로부터 유입되는 모든 신호를 무시하고 회전을 완수해야 한다. 회전 완료 후 정상 센서 폴링을 재개한다.

근거: 회전 중 센서 반응 시 무한 방향 재결정 루프가 발생할 수 있으므로 회전 완료를 원자 동작으로 처리한다.

1.3 청소 제어 (FR-CLEAN)

FR-CLEAN-01 – 표준 청소
전진 이동이 활성화된 상태에서 바닥 브러시와 물걸레 장치를 동시에 가동하여 청소를 수행해야 한다. 이동이 정지되면 청소 장치도 함께 정지한다.

FR-CLEAN-02 – 강화 청소 모드
먼지 센서에서 먼지가 감지되면 시스템은 100ms 이내에 청소 장치를 강화 모드(출력 증가)로 전환하고 5초 타이머를 시작해야 한다. 타이머 만료 시 자동으로 표준 모드로 복귀한다.

1.4 센서 입력 처리 (FR-SENSE)

FR-SENSE-01 – 장애물 감지
시스템은 청소 루프 반복마다 전방 근접 센서를 폴링하여 장애물 여부를 확인해야 한다. 전방 장애물 감지 시 FR-MOVE-02를 즉시 트리거한다. 장애물 감지 처리는 먼지 감지 처리보다 항상 우선한다.

FR-SENSE-02 – 먼지 감지
시스템은 청소 루프 반복마다 먼지 센서를 폴링하여 먼지 여부를 확인해야 한다. 먼지 감지 시 FR-CLEAN-02를 트리거하되, 현재 FR-MOVE-02(장애물 회피)가 처리 중이면 먼지 이벤트는 보류하지 않고 폐기한다.

NFR

2.1 아키텍처 품질 (NFR-ARCH)

NFR-ARCH-01 – 하드웨어 추상화
구동 모터, 청소 장치, 각 센서(전방·좌·우·먼지) 는 순수 추상 C++ 클래스(인터페이스)로 정의되어야 한다. 애플리케이션 로직은 이 인터페이스만 참조하며 구체 구현에 직접 의존하지 않는다.

NFR-ARCH-02 – Stub 기반 단위 테스트 가능성
물리 하드웨어 없이도 모든 핵심 로직의 단위 테스트를 실행할 수 있어야 한다. 이를 위해 NFR-ARCH-01의 각 인터페이스에 대응하는 Stub 구현을 제공해야 한다.

NFR-ARCH-03 – 예외 처리 및 안전 종료 보장
시스템은 어느 계층에서 예외가 발생하더라도 FR-CTRL-03의 안전 종료 흐름이 실행됨을 보장하는 예외 처리 구조를 가져야 한다. 예외를 전파하지 않고 소멸시키는 코드는 허용하지 않는다.

2.2 응답 시간 (NFR-TIMING)

ID	이벤트	제한 시간	측정 기준
NFR-TIMING-01	전원 ON → 자동 청소 루프 첫 반복 시작	≤ 5,000ms	FR-CTRL-01 완료 시점
NFR-TIMING-02	전방 센서 장애물 감지 → 전진 이동 완전 정지	≤ 50ms	FR-MOVE-02 진입 시점
NFR-TIMING-03	장애물 감지 후 방향 결정 → 모터 명령 발생 (좌/우 회전)	≤ 500ms	결정 시점 기준
NFR-TIMING-04a	전·좌·우 모두 막힘 → 후진 시작	≤ 100ms	FR-MOVE-02 후진 경로 진입
NFR-TIMING-04b	후진 완료 → 회전 방향 결정	≤ 500ms	후진 정지 시점 기준
NFR-TIMING-04c	회전 방향 결정 → 모터 명령 발생	≤ 500ms	결정 시점 기준
NFR-TIMING-05	먼지 감지 → 강화 모드 활성화	≤ 100ms	FR-CLEAN-02 진입 시점
NFR-TIMING-06	강화 모드 지속 시간	5,000ms	타이머 시작 기준

2.3 빌드 환경 (NFR-BUILD)

NFR-BUILD-01 – 구현 언어·빌드 시스템
- 구현 언어: C++17 표준 (`-std=c++17`)
- 빌드 시스템: CMake ≥ 3.14
- 컴파일러: GCC/G++ on Ubuntu (WSL 포함)
- 테스트 프레임워크: GoogleTest
- 로컬 빌드와 CI 빌드 환경에서 동일한 CMake 버전으로 빌드가 성공해야 한다.

2.4 사용자 인터페이스 (NFR-UI)

NFR-UI-01 – 실시간 상태 로그
시스템의 모드 전환(전진 시작, 회전, 후진, 강화 모드 활성/해제, 종료 등)은 발생 즉시 표준 출력에 텍스트 로그로 기록되어야 한다.

NFR-UI-02 – CLI 전원 제어
사용자는 터미널 CLI를 통해 전원 ON/OFF 명령을 입력할 수 있어야 한다.

2.5 동작 안전 규칙 (NFR-SAFETY)

NFR-SAFETY-01 – 이벤트 처리 우선순위
장애물 감지 이벤트는 먼지 감지 이벤트보다 항상 높은 우선순위를 갖는다. 두 이벤트가 같은 루프 반복에서 발생하면 장애물 처리가 먼저 실행되며, 해당 반복에서 먼지 이벤트는 폐기된다.

UC

UC-01: 시스템 기동

항목	내용
트리거	사용자가 CLI에서 전원 켜기 명령 입력
사전 상태	시스템이 오프라인(M-IDLE) 상태
주요 흐름	1. 시스템이 내부 구성 요소를 순차 초기화한다. ② 초기화 완료 후 준비 완료 메시지를 터미널에 출력한다. ③ UC-02 자동 청소로 자동 전이한다.
분기 / 예외	초기화 중 오류 발생 → UC-06 오류 종료로 전이
검증 시그널	전원 ON 명령 수신 후 5초 이내에 "준비 완료" 로그 출력 + UC-02 루프 진입

관련 FR: FR-CTRL-01, NFR-TIMING-01

UC-02: 자동 청소

항목	내용
트리거	UC-01 완료 후 자동 전이; 또는 UC-03 장애물 회피 완료 후 복귀
사전 상태	시스템이 청소 비활성 상태 (M-INIT 또는 M-AVOID 완료)
주요 흐름	① 청소 장치를 표준 강도로 활성화한다. ② 전진 이동을 시작한다. ③ [루프] 전방 센서 폴링 → 먼지 센서 폴링 → 반복.
분기 / 예외	전방 장애물 감지 → UC-03 전이 / 먼지 감지 → UC-04 전이 (단, 장애물 이벤트 우선) / 전원 OFF → UC-05 / 오류 → UC-06
검증 시그널	"전진 이동 시작" 로그 + 청소 장치 활성 상태 확인

관련 FR: FR-MOVE-01, FR-CLEAN-01, FR-SENSE-01, FR-SENSE-02, NFR-SAFETY-01

UC-03: 장애물 회피

항목	내용
트리거	전방 센서에서 장애물 감지 신호 수신 (UC-02 루프 중)
사전 상태	전진 이동 중 (M-CLEAN)
주요 흐름	① 전진 이동 및 청소 장치를 즉시 정지한다. ② 좌·우 센서를 조회하여 방향을 결정한다. ③ 결정된 동작(회전 또는 후진+회전)을 실행한다. ④ UC-02로 복귀한다.

UC-04: 강화 청소

항목	내용
트리거	먼지 센서에서 먼지 감지 신호 수신 (자동 청소 중; 장애물 회피 중이 아닐 것)
사전 상태	M-CLEAN (자동 청소 중), 강화 모드 비활성
주요 흐름	① 청소 장치를 강화 강도로 전환한다. ② 5초 타이머를 시작한다. ③ 타이머 만료 시 표준 강도로 복귀한다.
분기 / 예외	강화 모드 중 먼지 재감지 → 무시 (타이머 재시작 없음) / 강화 모드 중 장애물 감지 → UC-03 수행 후 강화 모드 유지하여 복귀 / 오류 → UC-06
검증 시그널	"강화 청소 활성" 로그 출력 후 5초 경과 시 "표준 청소 복귀" 로그 출력

관련 FR: FR-CLEAN-02, FR-SENSE-02, NFR-TIMING-05, NFR-TIMING-06

UC-05: 정상 종료

항목	내용
트리거	사용자가 CLI에서 전원 끄기 명령 입력
사전 상태	시스템이 활성 상태 (M-CLEAN, M-AVOID, M-BOOST 중 어느 하나)
주요 흐름	① 현재 진행 중인 이동·청소 동작을 중지한다. ② 구동 모터와 청소 장치를 정지한다. ③ "종료 중" 메시지를 출력한다. ④ 오프라인 상태(M-IDLE)로 전환한다.
분기 / 예외	종료 처리 중 오류 → UC-06 전이
검증 시그널	"시스템 종료" 로그 출력 + 모든 구동장치 정지 상태 확인

관련 FR: FR-CTRL-02

UC-06: 오류 종료

항목	내용
트리거	시스템 내부의 어느 단계에서든 복구 불가 예외/오류 발생 (UC-01~UC-05에서 전이 가능)
사전 상태	시스템 활성 상태 임의
주요 흐름	① 오류 내용을 터미널에 즉시 출력한다. ② 이동·청소 동작을 중지한다. ③ 구동 모터와 청소 장치를 정지한다. ④ 오프라인 상태로 전환한다.
분기 / 예외	없음 (최종 폴백 – 추가 예외 처리 없음)
검증 시그널	오류 메시지 로그 출력 + 모든 구동장치 정지 확인

관련 FR: FR-CTRL-03

추적성 표

원본 ID (PDF)	새 ID	흡수 방식
R1.1	FR-CTRL-01	재진술 (초기화 + 청소 루프 진입 통합)
R2.1	FR-MOVE-01, FR-CLEAN-01	이동 기능과 청소 기능으로 분리
R3.1	FR-SENSE-01	재진술
R3.2	FR-MOVE-02 사전 조건	FR-MOVE-02에 명시화 (별도 FR 불필요)
R3.3	FR-MOVE-02 (경로 1: 좌회전)	통합 (방향 결정 표의 첫 번째 행)
R3.4	FR-MOVE-02 (경로 2: 우회전)	통합 (방향 결정 표의 두 번째 행)
R3.5	FR-MOVE-02 (경로 3: 후진+회전)	통합 (방향 결정 표의 세 번째 행)
R4.1	FR-SENSE-02	재진술
R4.2	FR-CLEAN-02 (활성화 조건)	재진술
R4.3	FR-CLEAN-02 (지속 조건)	FR-CLEAN-02 내 부속 규칙으로 명시화
R5.1	FR-CTRL-02	재진술
R6.1	FR-CTRL-03	재진술

NFR-P-01	NFR-TIMING-01	재진술
NFR-P-02	NFR-TIMING-02	재진술; FR-MOVE-02 검증 기준에도 부속
NFR-P-03	NFR-TIMING-03	재진술
NFR-P-04	NFR-TIMING-04a·04b·04c	3단계로 분해하여 각각 측정 기준 명시
NFR-P-05	NFR-TIMING-05	재진술; FR-CLEAN-02 검증 기준에 부속
NFR-P-06	NFR-TIMING-06	FR-CLEAN-02 지속 시간 조건으로 명시화
NFR-O-01	NFR-ARCH-01	재진술 (추상 클래스 → 순수 추상 C++ 인터페이스로 명확화)
NFR-O-02	NFR-ARCH-02	재진술 (Stub 구현 명시)
NFR-O-03	NFR-ARCH-03	재진술 (예외 전파 금지 규칙 추가)
NFR-O-04	NFR-ARCH-03에 통합	입력 유효성 검증은 예외 처리 구조의 일부로 흡수
NFR-O-05	(삭제)	TP#5에서는 외부 서버·인증 정보가 없으므로 해당 없음; CI 환경 관련 내용은 objective.md 범위 밖
NFR-O-06	FR-MOVE-03	FR에 직접 명시화 (회전 중 입력 마스크 규칙)
NFR-O-07	FR-CLEAN-02 부속 규칙	강화 모드 중 먼지 재감지 무시 규칙으로 명시화
NFR-O-08	FR-CLEAN-02 부속 규칙	강화 모드 중 장애물 회피 허용 규칙으로 명시화
NFR-O-09	NFR-SAFETY-01	별도 안전 규칙으로 격상; FR-SENSE-01에도 우선순위 명시
NFR-OE-01	NFR-BUILD-01	재진술
NFR-I-01	NFR-UI-01	재진술
NFR-I-02	NFR-UI-02	재진술

/extract-specs 중 기록

주요 결정

1. 통합: 장애물 회피 UC4·UC5·UC6 → UC-03 단일 UC

원본의 UC3(감시 여부)·UC4(좌회전)·UC5(우회전)·UC6(우선+회전)은 공통 트리거(전방 장애물)에서 파생되는 세 분기 경로로, 독립 트리거가 없다. 이를 UC-03 하나로 통합하고 방향 결정 로직을 표로 표현하였다. 테스트 시나리오 작성 시 3개 경로를 표의 각 행으로 직접 대응시킬 수 있어 검증 가능성이 향상된다.

2. 변환: 타이밍 NFR → FR 검증 기준으로 이전

원본 NFR-P-01~P-06은 독립 항목으로 나열되어 있으나, 각 수치는 대응하는 기능 요구사항의 검증 기준으로 표현하는 것이 더 명료하다. NFR-TIMING-* 테이블을 별도로 유지하되 FR-CTRL-01, FR-MOVE-02, FR-CLEAN-02에 검증 기준으로도 부속시켰다.

3. 명시화: 입력 마스크 규칙을 별도 FR로 격상

원본 NFR-O-06(회전 중 센서 무시), NFR-O-07(강화 모드 중 먼지 무시), NFR-O-08(강화 모드 중 장애물 회피 허용)은 관련 FR(FR-MOVE-03, FR-CLEAN-02)의 부속 규칙으로 직접 명시화하였다. 별도 NFR로 분리하면 FR과 NFR 사이에 이중 기술이 생기고 추적성이 낮아진다.

4. 변환: NFR-O-09 → NFR-SAFETY-01로 격상

원본 NFR-O-09(장애물 > 먼지 우선순위)는 "Other requirements"에 묻혀 있었으나, 이 규칙은 이벤트 처리 순서를 결정하는 핵심 안전 정책이다. NFR-SAFETY-01로 격상하고 FR-SENSE-01에도 우선순위 언급을 추가하였다.

5. 삭제: NFR-O-05 (인증 정보 하드코딩 금지)

해당 요구사항은 CI/CD 환경의 AWS/GitHub Secrets 관리 정책으로, TP#5(로컬 WSL 빌드)에는 적용 대상이 없다. reference/ 정보에 따르면 이전 학기 팀의 CI/CD 인프라에 특화된 항목으로 판단하여 제외하였다.

6. 제외: PDF Domain Model·Class Diagram·Sequence Diagram

CLAUDE.md 2절·6절에 따라 PDF의 클래스 이름(RVCOrchestrator, CLIHandler 등), 메서드 시그니처(startCleaning(), requestStatus() 등), CICD-Team7-main 디렉토리 구조를 산출물에 일절 반영하지 않았다. 모든 FR/UC/NFR은 시스템의 의도(외부 자극과 기대 동작)만을 재진술한 것이다.

SRS 생성

- /srs 호출 시 참조되는 md

```
# /srs — Software Requirements Specification 작성

> **정책 (CLAUDE.md 2·6절)**
> 본 단계는 `/extract-specs`가 만든 **재증류된 명세**를 SRS 형식으로 재구성하는 단계다.
> PDF·이전 코드·원본 요구사항 ID는 다시 보지 않는다. 클래스 구조·메서드명은 SRS에서 결정하지 않는다 (그건 `/sdd`의 몫).
```

```
## 3. 작성 규칙

- **재증류된 새 ID만 사용**한다. 원본 PDF ID(R*, NFR-O-*, UC1~9)는 본문 어디에도 등장하면 안 된다.
- "구현 방법"을 적지 않는다 — "무엇을, 왜"만 적는다. 클래스명·메서드 시그니처·계층 구조는 본 단계에서 결정하지 않는다.
- 모든 표/다이어그램은 마크다운/mermaid로 인라인.
- mermaid는 **context diagram 수준만**. 도메인 클래스 다이어그램은 `/sdd`에서 그린다.
- 외부 의존성을 새로 도입할 만한 요구는 명시적으로 표시.
```

```
## 4. 자기 점검

- [ ] PDF 원본 ID(R*, NFR-O-* 등)가 SRS 본문에 등장하지 않는가?
- [ ] PDF Domain Model 클래스명(RVCOrchestrator, MovementPolicyController 등)이 SRS 본문에 등장하지 않는가?
- [ ] FR/NFR/UC가 모두 `docs/specs/`의 ID 체계와 일치하는가?
- [ ] `reference/` 폴더를 본 단계에서 한 번도 읽지 않았는가?
```

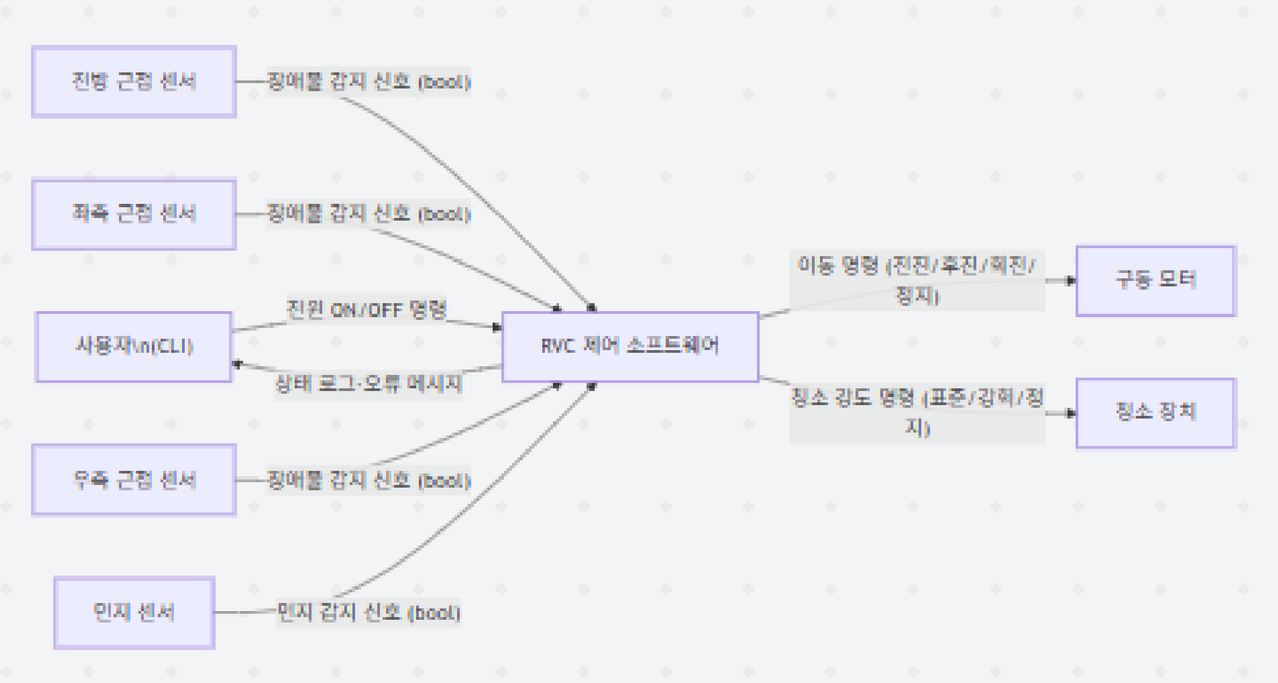
```
## 2. 산출물

`docs/srs/SRS.md` — 다음 섹션 구조 필수:

1. 문서 개요 (목적, 범위, 정의·약어, 참고 문서 — `docs/specs/`만 명시)
2. 시스템 개요
   - 시스템 컨텍스트 (mermaid context diagram — `system-overview.md`를 SRS 형식으로 재배치)
   - 외부 액터 (재증류된 명세에 정의된 액터만)
3. 기능 요구사항 (FR)
   - `docs/specs/requirements.md`의 FR을 SRS 표 형식으로 옮긴다 (ID 유지)
   - 각 FR은 검증 가능한 진술 + 측정 기준 포함
4. 비기능 요구사항 (NFR)
   - `docs/specs/requirements.md`의 NFR을 카테고리별로 재배치 (성능·아키텍처·빌드·인터페이스 등)
5. 유스케이스 (UC)
   - `docs/specs/use-cases.md`의 UC를 SRS 표준 양식(ID/이름/액터/사전조건/주요 흐름/대안·예외/사후조건)으로 재배치
6. 외부 인터페이스
   - 사용자 인터페이스: 재증류된 명세의 CLI 요구만
   - 하드웨어 추상화: "외부 환경과의 경계가 추상화되어야 한다"는 수준의 요구 (구체 인터페이스 이름·시그니처는 적지 않음 — `/sdd`가 결정)
7. 추적성 매트릭스
   - 재증류된 FR ↔ UC 매핑 (다음 단계 `/sdd`가 "컴포넌트" 열을 채우도록 빈 칸 유지)
```

SRS

- SRS.md
  - 시스템 컨텍스트




```
## 자기 점검 결과

- [x] PDF 원본 ID(R*, NFR-O-* 등)가 SRS 본문에 등장하지 않는다.
- [x] PDF Domain Model 클래스명(RVCOrchestrator, MovementPolicyController 등)이 SRS 본문에 등장하지 않는다.
- [x] FR/NFR/UC가 모두 `docs/specs/`의 ID 체계와 일치한다.
- [x] `reference/` 폴더를 본 단계에서 한 번도 읽지 않았다.
```

/srs 중 기록

주요 결정

1. FR → SRS 표 형식 재배치: "조건→동작→사후 상태" 구조 명료화
`docs/specs/requirements.md`의 FR은 문단 형식으로 기술되어 있었다. SRS에서는 각 FR을 ID·요구사항·사전조건·사후조건·측정기준·우선순위의 6-항목 표로 재배치하였다. 이를 통해 단위 테스트 및 시스템 테스트 작성 시 어떤 상태에서 무엇을 확인해야 하는지 즉시 파악 가능하다.

2. NFR-TIMING을 본문 표로 중앙화 + FR 섹션에 교차 참조 유지
NFR-TIMING-01~06+04a~c를 `§4.2`에 단일 표로 정리하고, FR-MOVE-02 및 FR-CLEAN-02의 "측정 기준" 셀에 NFR 참조 번호를 함께 기재하였다. 두 곳을 동기화하면 사양이 바뀌어도 추적이 끊기지 않는다.

3. UC를 SRS 표준 양식(6항목)으로 재배치
`docs/specs/use-cases.md`의 UC는 5요소(트리거·사전 상태·주요 흐름·분기/예외·검증 시그널)로 구성되어 있었다. SRS에서는 UC ID/명칭/주 액터/목적/사전조건/주요 흐름/대안·예외/사후조건/검증 시그널/관련 FR의 9항목 표로 확장하였다. "주 액터"를 추가하여 각 UC가 누구에 의해 트리거되는지 명확히 하였다.

4. 추적성 매트릭스: "컴포넌트" 열을 빈 칸으로 유지
FR/NFR ↔ UC 매핑을 완성하되, `/sdd` 단계에서 채울 "컴포넌트" 열을 의도적으로 빈 칸으로 남겼다. 이 설계는 SRS가 "무엇을"만 답하고 "어떻게"는 SDD에 위임한다는 원칙을 유지한다.

5. 외부 인터페이스 섹션: 추상화 경계만 명세, 구체 시그니처 제외
하드웨어 추상화 경계를 "입력/출력 방향 + 신호 타입" 수준으로만 기술하고, 인터페이스 이름·메서드 시그니처는 기재하지 않았다. 클래스명·메서드명 결정은 `/sdd`의 책임이다.

SDD 생성

- /sdd 호출 시 참조되는 md

```
# /sdd — Software Design Document 작성

> **정책 (CLAUDE.md 2·6절)**
> 본 단계가 RVC의 **설계(How)**를 처음 결정한다.
> PDF·이전 코드의 Domain Model·Sequence Diagram·메서드 시그니처를 답습하지 않는다. 클래스 분해·인터페이스 추상화·계층 구조는 SRS의 요구사항으로부터 Claude가 직접 도출한다.
```

```
## 2. 산출물

`docs/sdd/SDD.md` — 필수 섹션 (내용은 Claude가 SRS에서 도출):

1. **설계 개요**
   - 설계 목표 (SRS NFR 충족 기준)
   - 설계 원칙 (SOLID, 의존성 역전, 테스트 용이성, 단일 책임)
2. **아키텍처 결정**
   - 계층 구조를 Claude가 명명하고 그린다 (mermaid). 계층 수·이름·역할은 SRS 요구사항에서 직접 도출 — PDF Domain Model의 6 컨트롤러 분해를 강제 채택하지 않는다.
   - 결정 근거: 왜 이 분해인가, 어떤 NFR을 충족시키는가
3. **컴포넌트 설계**
   - 각 컴포넌트(클래스/모듈)마다: 책임(한 문장), 의존하는 추상화, 공개 메서드 시그니처(C++17), 상태 관리 방식
   - 컴포넌트 수와 명칭은 Claude가 결정 — PDF의 `RVCOrchestrator`·`MotorController`·`MovementPolicyController` 같은 이름을 그대로 채택해서는 안 된다 (의도적으로 다른 이름·다른 분해 권장)
4. **추상화 인터페이스**
   - 외부 환경(센서·구동기·사용자 입출력 등)을 추상화하는 인터페이스 명세
   - 인터페이스명·메서드명 모두 Claude가 결정 — PDF Sequence Diagram의 `startCleaning()`·`requestStatus()` 등 메시지명을 그대로 받아쓰지 않는다
```

```
5. **상태·이벤트 모델**
   - 시스템 운영 모드와 전이를 다이어그램으로 표현 (mermaid stateDiagram)
   - SRS의 UC 흐름과 일치해야 함
6. **시퀀스 설계** (mermaid sequenceDiagram)
   - SRS UC 중 핵심 3~5개에 대한 시퀀스
   - 등장 메시지명은 Claude가 SRS 요구에서 새로 정의 (PDF 다이어그램 메시지 그대로 재현 금지)
7. **데이터·타입 설계**
   - enum, 값 타입, 공용 타입 헤더 위치
   - 타입 이름은 Claude 결정 — `DirectionType`·`StateType`·`ErrorType` 같은 PDF 이름의 답습을 피한다
8. **디렉토리·파일 매핑**
   - 어떤 컴포넌트가 어느 `include/<sub>/`, `src/<sub>/` 파일에 들어갈지 표
   - 디렉토리 구조도 Claude 결정 — `controller/`·`handler/`·`hardware/` 같은 분류가 자연스럽다면 채택하되, 더 적합한 구조가 있으면 그것을 택한다
9. **빌드 설계**
   - CMake 타겟 정의 (메인 실행파일 + 단위 테스트 실행파일 + 시스템 테스트 실행파일)
   - 의존성 그래프 (rvc_core 라이브러리 + 실행파일 분리 권장, 단 Claude가 더 나은 구조를 도출하면 그것 채택)
10. **테스트 가능성 보장**
    - 모든 컴포넌트를 Stub으로 격리해 단위 테스트 가능함을 설계 차원에서 보장 (SRS NFR 충족)
    - 출력 사이드 이펙트(`std::cout` 등)는 단일 책임 통제점을 거쳐야 함
11. **SRS ↔ 컴포넌트 추적성 매트릭스**
    - SRS의 FR-ID ↔ Claude가 새로 정의한 컴포넌트/메서드 매핑
```

```
## 4. 자기 점검

- [ ] PDF Domain Model의 클래스명(RVCOrchestrator, MovementPolicyController, CleaningPolicyController, RVCPowerController, MotorController, CleanerController, ErrorHandler 등)이 SDD에 그대로 등장하지 않는가? (의도적으로 다른 이름을 선택했는가?)
- [ ] PDF Sequence Diagram의 메시지명(`startCleaning()`, `requestStatus()`, `moveForward()`, `powerUp()`, `stopMoving()`, `stopCleaning()`, `shutdown()`, `turnLeft()`, `turnRight()`, `moveBackward()` 등)이 SDD 인터페이스 시그니처로 그대로 등장하지 않는가?
- [ ] 모든 SRS FR이 추적성 매트릭스에서 어떤 컴포넌트에 매핑되어 있는가?
- [ ] 모든 SRS NFR이 설계의 어느 측면에서 충족되는지 명시되어 있는가?
- [ ] `reference/`·`docs/specs/`를 본 단계에서 한 번도 읽지 않았는가?
```

SDD

- 작성된 sdd.md

```
## 1. 설계 개요

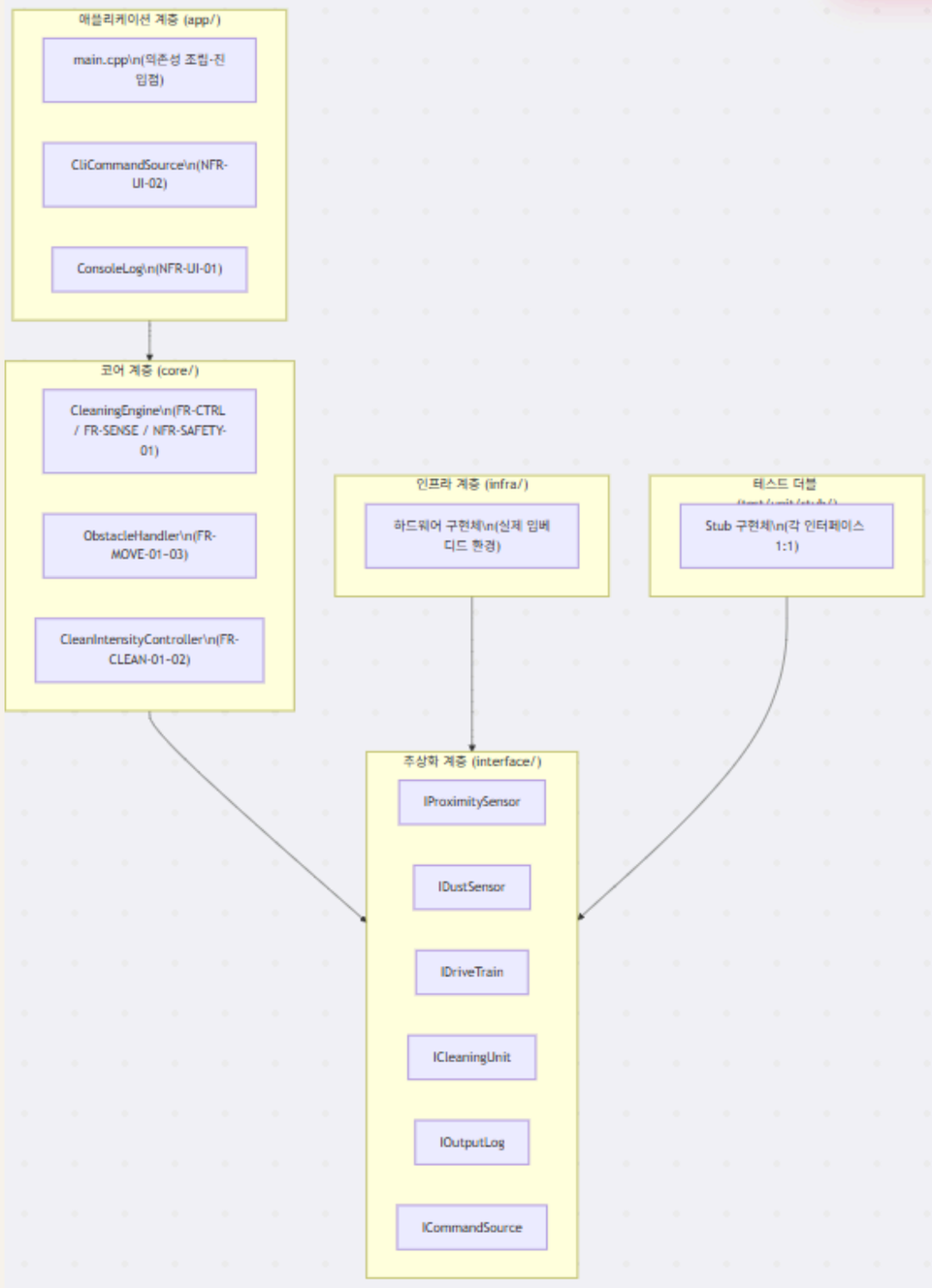
### 1.1 설계 목표

| SRS 요구사항 | 충족 설계 전략 |
|-------------|---------------|
| NFR-ARCH-01: 하드웨어 추상화 | 모든 물리 장치를 순수 가상 C++ 인터페이스로 분리; 코어 로직은 인터페이스만 참조 |
| NFR-ARCH-02: Stub 기반 테스트 | 각 인터페이스에 1:1 대응하는 Stub을 `test/unit/stub/`에 제공 |
| NFR-ARCH-03: 예외 안전 종료 | `CleaningEngine` 최상위에서 모든 예외를 포착하여 FR-CTRL-03 경로 실행 보장 |
| NFR-TIMING-02: 장애물→정지 ≤50ms | 이벤트 루프 주기를 50ms 미만으로 유지; 모터 정지 명령이 루프 내 최우선 처리 |
| NFR-SAFETY-01: 장애물 우선 | 루프 내 센서 폴링 순서를 장애물→먼지로 고정; 동일 반복에서 장애물 처리 시 먼지 이벤트 폐기 |
| NFR-BUILD-01: C++17/CMake 3.14 | `target_compile_features(... cxx_std_17)`, CMake minimum_required(3.14) |

### 1.2 설계 원칙

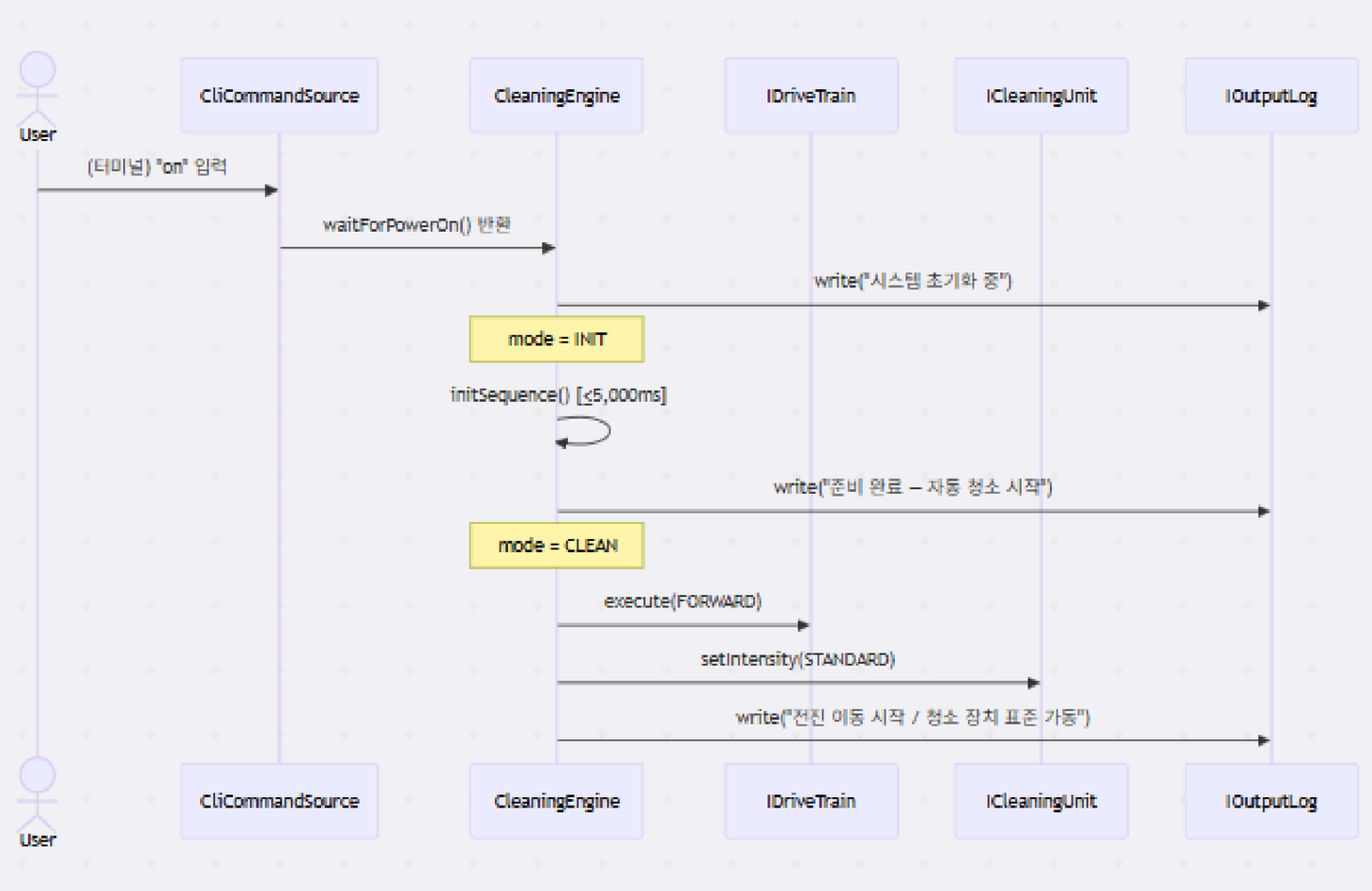
- **단일 책임(SRP)**: 각 클래스는 하나의 SRS 요구 그룹(FR-CTRL, FR-MOVE, FR-CLEAN, FR-SENSE)에만 책임을 진다.
- **의존성 역전(DIP)**: 코어 로직은 구체 하드웨어 클래스가 아닌 인터페이스(`I*`)에만 의존한다.
- **인터페이스 분리(ISP)**: 각 인터페이스는 단일 책임의 최소 메서드만 선언한다.
- **개방·폐쇄(OCP)**: 새 센서/구동기 추가는 인터페이스 구현 추가로만 처리하며 코어 코드를 변경하지 않는다.
- **테스트 용이성**: 의존성 주입(생성자 주입)을 통해 Stub 교체가 가능하도록 설계한다.
```

- 상태 전이 다이어그램



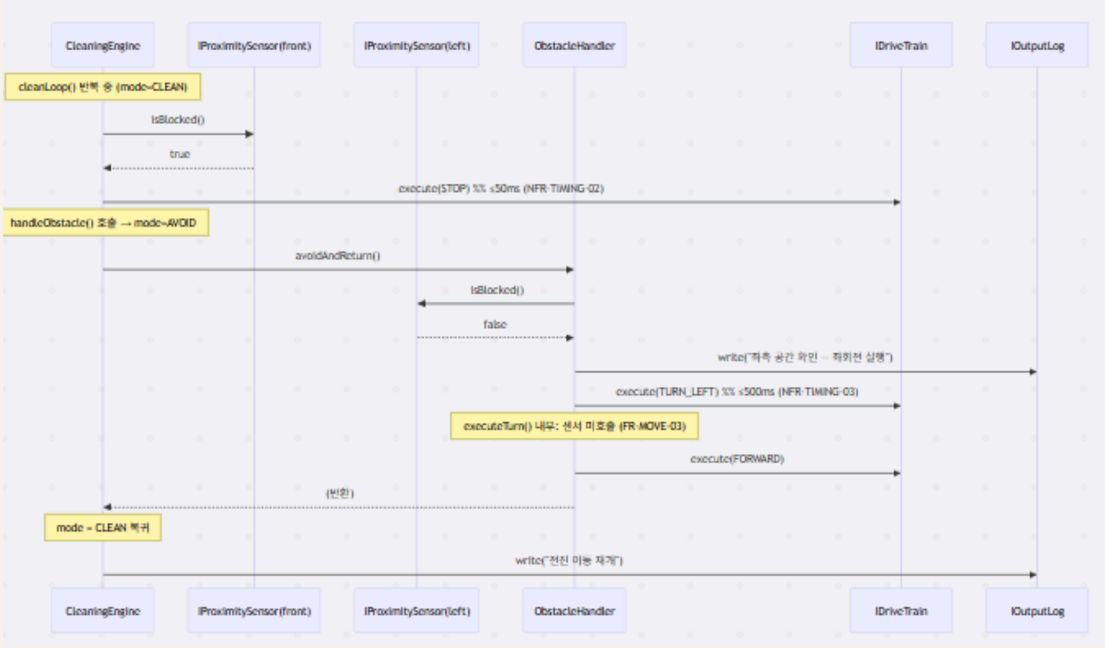
SDD

- UC-01: 시스템 가동



SDD

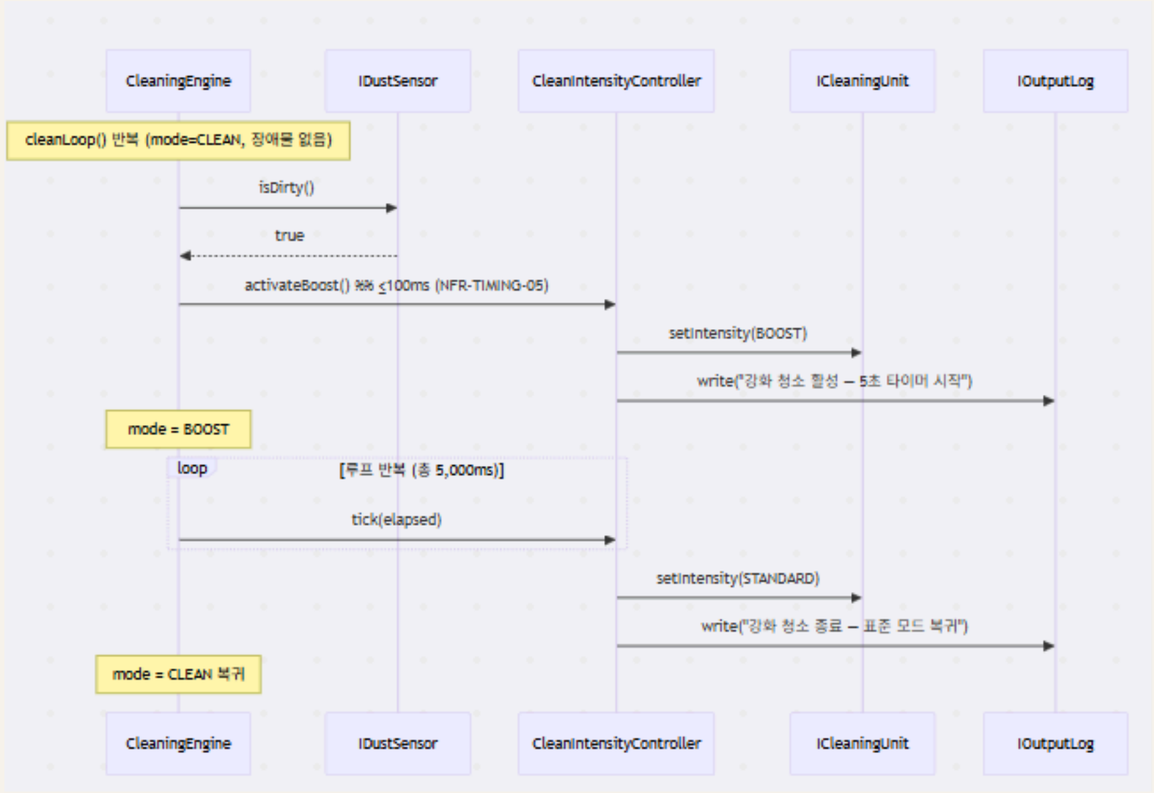
- UC-03: 장애물 회피 – 좌회전 경로



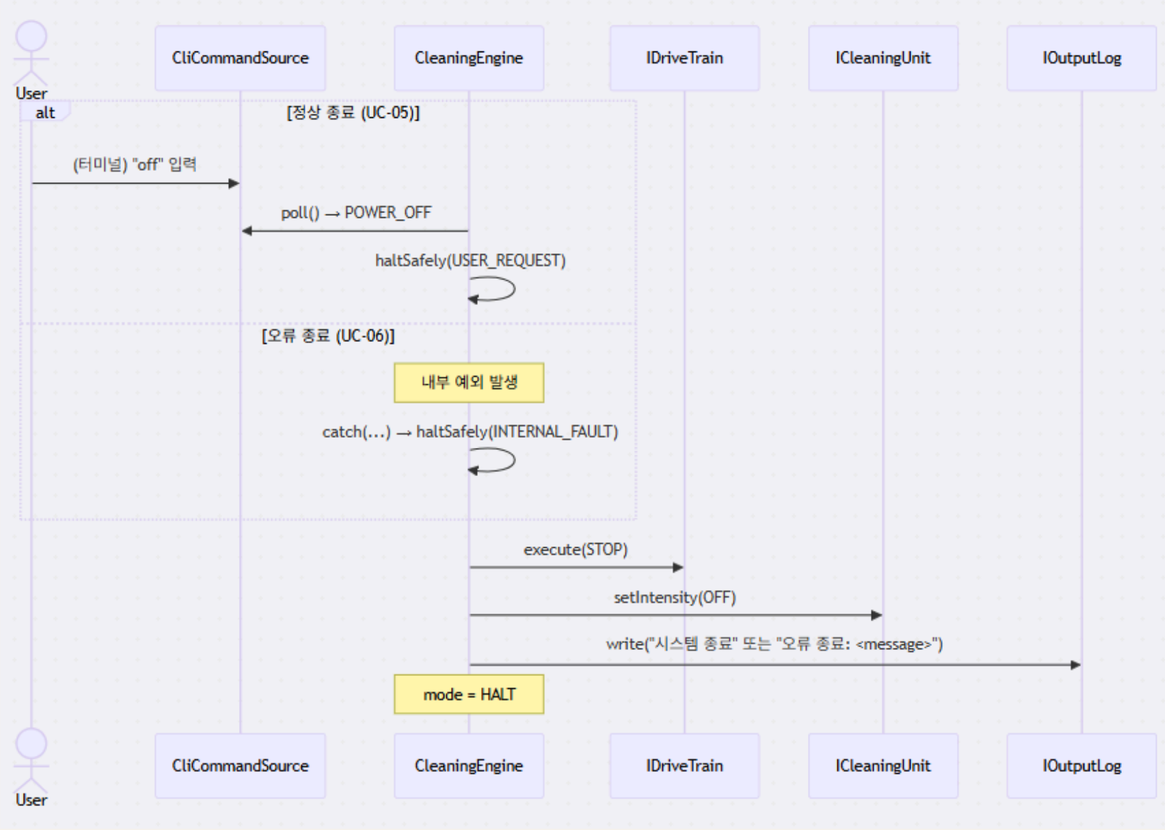
- UC-03: 장애물 회피 – 후진+재조회 경로

SDD

- UC-04: 강화 청소



- UC-05/UC-06: 정상 오류 종료



/sdd 중 기록

주요 결정

1. 계층 구조: 4계층 분리 (interface / core / app / infra)
SRS NFR-ARCH-01·02의 "하드웨어 추상화 + Stub 기반 테스트" 요구에서 직접 도출하였다. 추상화 계층(interface/)을 물리적으로 분리함으로써 코어 로직이 구체 하드웨어 파일을 절대 포함하지 않음을 파일 의존성 구조로 강제한다.

2. PDF Domain Model 대비 의도적 분해 차이 (발표 핵심)

PDF 분해	본 설계	변경 근거
`RVCOrchestrator`	`CleaningEngine`	오케스트레이터는 불필요한 간접층; FR-CTRL + 이벤트 루프를 하나에 집약
`MovementPolicyController`	`ObstacleHandler`	"Policy"는 책임이 모호; 회전 마스크(FR-MOVE-03)까지 포함한 핸들러
`CleaningPolicyController`	`CleanIntensityController`	청소 정책 = 강도 제어 + 타이머; 이름에서 역할을 즉시 파악
`RVCPowerController`	`CleaningEngine` 내부 흡수	전원 제어 = 생명주기 = 루프 제어; 별도 클래스는 SRP 위반
`MotorController` (구현)	`IDriveTrain` + 인프라 구현	인터페이스/구현 분리 철저화
`FrontSensor`, `LeftSensor`, `RightSensor` (각 서브클래스)	`IProximitySensor` × 3 인스턴스	계약이 동일 — 별도 인터페이스는 ISP 위반; 위치는 인스턴스로 구분
`ErrorHandler`	`CleaningEngine`의 `try-catch`	오류 처리 = 루프 최외곽 안전망; 별도 클래스는 예외 흐름 복잡화

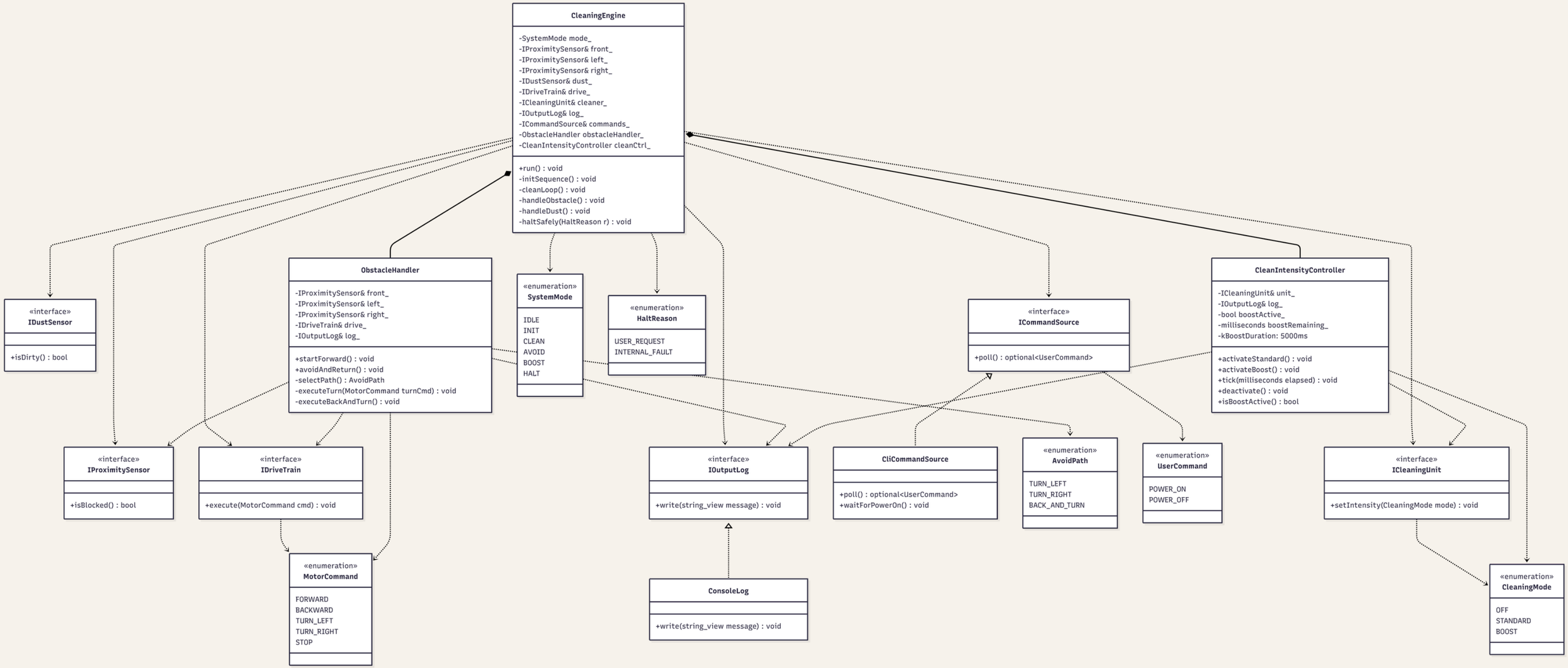
3. 인터페이스 메서드명: 단일 execute/setIntensity 패턴
PDF의 `moveForward()`, `turnLeft()`, `startCleaning()` 등 다수의 개별 메서드 대신 `IDriveTrain::execute(MotorCommand)`와 `ICleaningUnit::setIntensity(CleaningMode)`로 통합. 새 명령·모드 추가 시 열거자만 확장하면 인터페이스 변경 불필요(OCP).

4. Stub 격리 방식
`StubDriveTrain::commandAt(i)` 등 기록 기반 API — "어떤 순서로 어떤 명령이 발생했는지" 단위 테스트에서 정밀 단언. 회전 중 센서 미호출(FR-MOVE-03)은 `StubProximitySensor::callCount()`로 검증.

5. 출력 사이드 이펙트 통제점: IOutputLog
`std::cout`을 `IOutputLog::write()` 뒤에 감춤. 코어 로직 파일에 `<iostream>` include 없음. 단위 테스트에서 `StubOutputLog`로 교체해 로그 내용을 단언 가능.

6. 타이머 테스트 가능성: tick(elapsed) 주입
`CleanIntensityController::tick(ms elapsed)` — 경과 시간을 외부 인수로 받으므로 테스트에서 `tick(5001ms)` 한 번으로 5초를 즉시 시뮬레이션.

Class Diagram



Code 생성

- /code 시 호출되는 md

```
# /code — C++ 구현

> **정책 (CLAUDE.md 2·6절)**
> 구현은 **SDD를 진실 소스**로 한다. SDD가 정의하지 않은 클래스/파일을 임의로 만들지 않는다.
> PDF·이전 학기 코드(`reference/CICD-Team7-main/`)의 디렉토리 구조·헤더 이름을 답습하지 않는다 — 본 단계의 입력에 포함되지 않는다.
```

```
## 1. 입력 (이외 자료 보지 말 것)

- `docs/sdd/SDD.md` (단일 진실 소스 — 디렉토리·파일·클래스명·시그니처 모두 여기서 따옴)
- `docs/srs/SRS.md` (사후조건·검증 기준 확인용)

`reference/`, `docs/specs/`는 본 단계에서 열지 않는다.
```

```
## 3. 구현 규칙

- C++17 표준 엄수, `-Wall -Wextra -Wpedantic` 경고 0.
- 헤더 가드: `#pragma once`.
- RAII, smart pointer 우선. 원시 포인터는 비소유 의존성에 한해 허용.
- 의존성 주입: 생성자로 추상화 포인터/참조 받기. 구체 클래스 간 직접 결합 금지.
- 표준 출력 메시지는 SDD가 정한 단일 책임 통제점을 통해서만.
- **PDF/이전 코드의 클래스명·메서드명을 우연이라도 그대로 가져오지 않는다.** SDD가 정의한 이름만 사용.

## 4. 빌드 검증 (WSL)

```bash
wsl bash -lc "cd '$(pwd)' && cmake -S . -B build && cmake --build build -j 2>&1 | tee /tmp/rvc_build.log"
```

빌드 실패 시 즉시 수정. 경고도 0이어야 함. 검증이 끝날 때까지 본 단계는 완료가 아니다.

빌드 오류가 SDD의 모호함에서 비롯되면, 코드를 임시방편으로 고치지 말고 `/sdd`로 돌아가 SDD를 보강한 뒤 다시 `/code`를 진행한다.
```

```
## 5. 자기 점검

- [ ] 모든 소스·헤더 파일이 SDD 8절 매핑 표에 등장하는가? (SDD에 없는 임의 파일을 만들지 않았는가?)
- [ ] 모든 클래스명·메서드명·인터페이스명이 SDD에 정의된 그대로인가?
- [ ] PDF Domain Model/Sequence Diagram의 이름(RVCOrchestrator·MovementPolicyController·startCleaning·moveForward 등)을 우연히 차용하지 않았는가?
- [ ] 빌드 경고 0개, 모든 단위 테스트가 (있다면) 통과하는가?
- [ ] `reference/`·`docs/specs/`를 본 단계에서 한 번도 열지 않았는가?
```

Code

주요 결정

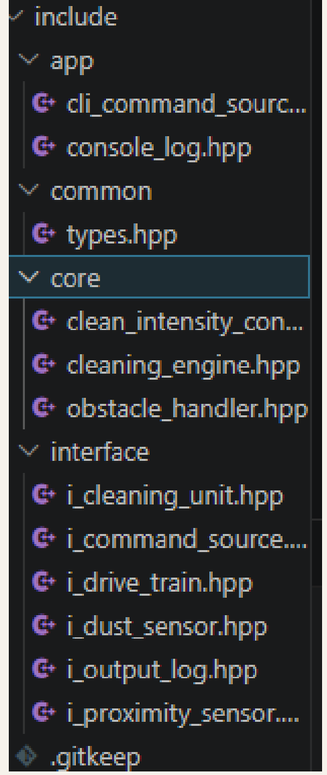
1. 의존성 주입: 생성자 레퍼런스 주입
모든 인터페이스 의존성은 생성자에서 참조(`&`)로 받는다. `CleaningEngine`은 `ObstacleHandler`와 `CleanIntensityController`를 멤버로 소유하며, 이들도 동일한 인터페이스 레퍼런스를 받는다. 원시 포인터·스마트 포인터 없이 소유권이 명확하다.

2. SDD 보강 항목 (구현 중 발견)
- `CleanIntensityController`에 `resume()` 메서드 추가 — 장애물 회피 후 청소 강도를 이전 상태(표준/강화)로 복원하기 위해 필요. SDD의 `deactivate()`는 하드웨어만 끄고 내부 부스트 상태를 보존하도록 구현함. 이로써 FR-CLEAN-02 "장애물 회피 후 강화 모드 유지 복귀"를 구현 가능.
- `CleaningEngine` 생성자에 `loopInterval` 파라미터 추가(기본 0ms) — 프로덕션은 100ms, 테스트는 0ms로 즉시 실행.
- `ObstacleHandler::executeTurn()`은 `drive.execute(turnCmd)` 직후 `drive.execute(STOP)` 호출 — 회전 완료 후 정지. 이 두 호출 사이에 센서를 호출하지 않아 FR-MOVE-03을 구현함.

3. 출력 사이드 이펙트 통제
`std::cout`은 `ConsoleLog::write()` 내부에만 존재. 코어 코드(`core/`, `interface/`) 전체에 `<iostream>` include 없음. 단위 테스트에서 `StubOutputLog`를 주입하면 콘솔 출력 없이 로그 내용을 단언 가능.

4. CMakeLists.txt: FetchContent 사용
SDD는 `find_package(GTest)` 방식을 명시했으나, `libgtest-dev`는 소스 패키지이므로 별도 컴파일이 필요하다. `FetchContent`가 더 안정적이고 재현 가능성이 높아 /code 템플릿 방식을 채택함.

5. 테스트 설계: StubCommandSource.enqueuePause()
`cleanLoop()`가 매 반복 최상단에서 `commands_.poll()`을 호출하므로, POWER_OFF 이전에 루프가 한 번 이상 실행되어야 하는 테스트에 `enqueuePause()`(명시적 nullopt)를 사용했다.



/code 중 기록

주요 결정

1. 의존성 주입: 생성자 레퍼런스 주입
모든 인터페이스 의존성은 생성자에서 참조(`&`)로 받는다. `CleaningEngine`은 `ObstacleHandler`와 `CleanIntensityController`를 멤버로 소유하며, 이들도 동일한 인터페이스 레퍼런스를 받는다. 원시 포인터·스마트 포인터 없이 소유권이 명확하다.

2. SDD 보강 항목 (구현 중 발견)
- `CleanIntensityController`에 `resume()` 메서드 추가 — 장애물 회피 후 청소 강도를 이전 상태(표준/강화)로 복원하기 위해 필요. SDD의 `deactivate()`는 하드웨어만 끄고 내부 부스트 상태를 보존하도록 구현함. 이로써 FR-CLEAN-02 "장애물 회피 후 강화 모드 유지 복귀"를 구현 가능.
- `CleaningEngine` 생성자에 `loopInterval` 파라미터 추가(기본 0ms) — 프로덕션은 100ms, 테스트는 0ms로 즉시 실행.
- `ObstacleHandler::executeTurn()`은 `drive.execute(turnCmd)` 직후 `drive.execute(STOP)` 호출 — 회전 완료 후 정지. 이 두 호출 사이에 센서를 호출하지 않아 FR-MOVE-03을 구현함.

3. 출력 사이드 이펙트 통제
`std::cout`은 `ConsoleLog::write()` 내부에만 존재. 코어 코드(`core/`, `interface/`) 전체에 `<iostream>` include 없음. 단위 테스트에서 `StubOutputLog`를 주입하면 콘솔 출력 없이 로그 내용을 단언 가능.

4. CMakeLists.txt: FetchContent 사용
SDD는 `find_package(GTest)` 방식을 명시했으나, `libgtest-dev`는 소스 패키지이므로 별도 컴파일이 필요하다. `FetchContent`가 더 안정적이고 재현 가능성이 높아 /code 템플릿 방식을 채택함.

5. 테스트 설계: StubCommandSource.enqueuePause()
`cleanLoop()`가 매 반복 최상단에서 `commands_.poll()`을 호출하므로, POWER_OFF 이전에 루프가 한 번 이상 실행되어야 하는 테스트에 `enqueuePause()`(명시적 nullopt)를 사용했다.

Unit Test 생성

- /ut 실행 시 참조하는 md

```
# /ut — Unit Tests

> **정책**
> 테스트 파일과 단위는 **실제 `include/`·`src/`에 존재하는 클래스**에 맞춰 도출한다.
> PDF Domain Model의 클래스 분해를 기준으로 테스트 파일 이름을 미리 정해두지 않는다.
```

```
## 4. 테스트 작성 규칙

- 각 컴포넌트 공개 메서드마다 **Positive + Negative** 최소 1쌍.
- 테스트 이름: `<Subject>_<Scenario>_<ExpectedOutcome>` (예: `<클래스>_<상황>_<기대결과>`).
- 출력 검증은 SDD가 정한 출력 통제점에 대한 의존성 주입을 통해 수행. `testing::internal::CaptureStdout()`은 최후의 수단.
- 시간 의존 코드는 의존성 주입으로 격리.
- 한 테스트가 한 가지만 검증. assertion 폭주 금지.
- 각 테스트 파일 상단 주석에 검증 대상 SRS FR/UC ID 명시 (SDD 추적성 매트릭스 활용).
```

```
## 2. 테스트 단위 도출

1. `include/`·`src/`를 훑어 **실제로 존재하는 테스트 가능한 클래스**(단일 책임을 가진 컴포넌트, 상태나 결정 로직을 보유한 것)를 식별.
2. 클래스마다 `test/unit/<ClassName>Test.cpp` 1개를 두는 것을 기본으로 한다. 파일명은 **실제 클래스명을 따른다** — PDF 이름을 가정해 미리 적어두지 않는다.
3. 외부 의존성(센서·모터 등)을 받는 컴포넌트는 Stub/Spy로 격리해 테스트.

## 3. 산출물

`test/unit/` 아래 GTest 기반 테스트 파일. 구조:

- 컴포넌트별 테스트 파일 (이름은 실제 클래스에 맞춰 결정)
- Stub/Spy 클래스는 `test/unit/stubs/` 또는 각 테스트 파일 내부에 두되, **production 코드(`src/`, `include/`)에 절대 두지 않는다**.
```

Unit Test

test > unit > stub > stub_cleaning_unit.hpp > ...

```cpp
#pragma once
#include <vector>
#include "interface/i_cleaning_unit.hpp"

class StubCleaningUnit final : public ICleaningUnit {
public:
    void setIntensity(CleaningMode mode) override {
        history_.push_back(mode);
    }

    CleaningMode lastMode() const { return history_.back(); }

    CleaningMode modeAt(std::size_t i) const { return history_.at(i); }

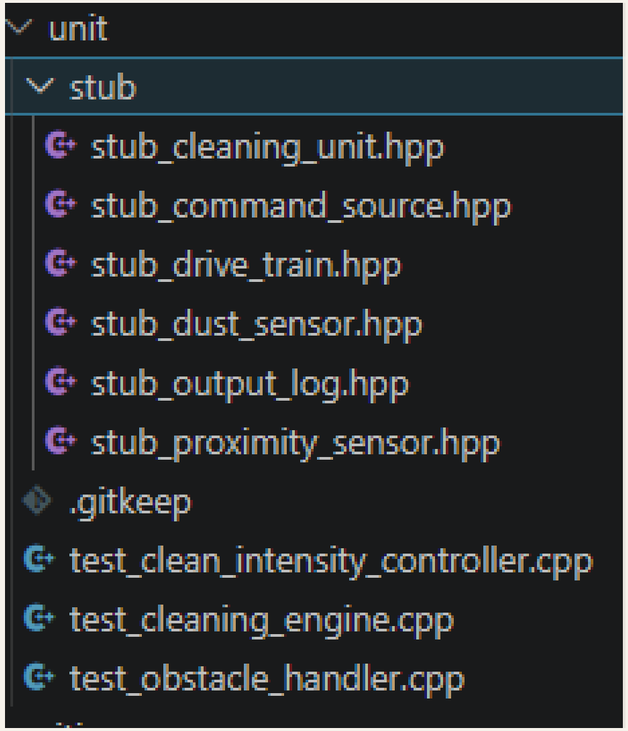
    bool hasMode(CleaningMode mode) const {
        for (auto& m : history_) if (m == mode) return true;
        return false;
    }

    std::size_t modeCount() const { return history_.size(); }

    void reset() { history_.clear(); }

private:
    std::vector<CleaningMode> history_;
};

```



/ut 실행 중 기록

주요 결정

1. Stub vs Mock 선택
모든 외부 의존성(센서·모터·청소기·로그·커맨드)에 **Stub** 클래스를 사용.
- `StubProximitySensor` / `StubDustSensor`: `enqueueResult(bool)` + `setBlocked(bool)` 이중 레이어 — 첫 N번 폴은 큐에서 뽑고, 이후는 `defaultBlocked_` 반환. 순서 의존 시나리오와 정상 상태 시나리오를 모두 처리.
- `StubDriveTrain`: `history_` 벡터로 전체 명령 시퀀스 보존 — `commandAt(i)`, `commandCount()`로 순서 검증 가능.
- `StubCommandSource.enqueuePause()`: 명시적 `std::nullopt` 주입으로 루프 N회 실행 후 POWER_OFF 처리.
- Mock 미사용 이유: 호출 순서 검증이 필요한 경우도 Stub의 history 벡터로 충분히 달성 가능하며, GoogleMock 의존 없이 헤더 단독으로 동작.

2. FR ↔ 테스트 매핑

FR/NFR ID	검증 테스트
FR-CTRL-01	`CleaningEngineTest.StartupStartsForwardAndStandardCleaning`, `Startup_LogsReadyMessage`
FR-CTRL-02	`CleaningEngineTest.PowerOffHaltsAllDevices`
FR-CTRL-03	`CleaningEngineTest.ExceptionCausesSafeHalt`, `SystemScenario.InternalErrorCausesSafeHalt`
FR-MOVE-01	`ObstacleHandlerTest.StartForwardEmitsForwardCommand`, `StartForward_EmitsExactlyOneForward`
FR-MOVE-02	`ObstacleHandlerTest.TurnLeft/Right/BackwardThen*`, `CommandOrder_LeftTurn_*`, `CommandOrder_BackPath_*`
FR-MOVE-03	`ObstacleHandlerTest.NoSensorPollingDuringRotation`
FR-CLEAN-01	`CleanIntensityControllerTest.ActivateStandardSetsStandardMode`, `ActivateStandard_AfterBoost_ClearsBoostFlag`
FR-CLEAN-02	`BoostExpiresAfter5000ms`, `Boost_ExpiresAtExactBoundary_5000ms`, `Boost_AccumulatesAcrossMultipleTicks`, `SecondActivateBoostDoesNotRestartTimer`, `DeactivatePreservesBoostStateForResume`, `DustIgnoredDuringBoostMode`, `ObstacleHandledDuringBoostMode_BoostResumesAfter`
FR-SENSE-01	`CleaningEngineTest.FrontObstacleTriggersAvoidance`, `SystemScenario.ObstacleAvoidance*`
FR-SENSE-02	`CleaningEngineTest.DustActivatesBoostMode`, `SystemScenario.BoostActivatedByDust`
NFR-SAFETY-01	`CleaningEngineTest.ObstaclePriorityOverDust`
NFR-TIMING-06	`BoostExpiresAfter5000ms`, `BoostDoesNotExpireBefore5000ms`, `Boost_ExpiresAtExactBoundary_5000ms`, `Boost_AccumulatesAcrossMultipleTicks`

3. 추가된 테스트 (10개) — 경계 케이스 및 시퀀스 검증

test_obstacle_handler.cpp (3개 추가):
- `CommandOrder_LeftTurn_TurnBeforeStopBeforeForward`: TURN_LEFT → STOP → FORWARD 정확한 순서 검증 (FR-MOVE-02 + FR-MOVE-03)
- `CommandOrder_BackPath_BackwardBeforeTurnBeforeForward`: 후진 경로에서 BACKWARD가 TURN 이전에 발행됨을 인덱스로 검증
- `StartForward_EmitsExactlyOneForward`: `startForward()`가 정확히 명령 1개만 발행하는지 검증

test_clean_intensity_controller.cpp (4개 추가):
- `Boost_ExpiresAtExactBoundary_5000ms`: `tick(5000ms)` — 정확히 경계에서 만료 (`boostRemaining_ <= 0` 조건)
- `Boost_AccumulatesAcrossMultipleTicks`: 2000ms + 2000ms + 1001ms = 5001ms 누적 만료
- `ActivateStandard_AfterBoost_ClearsBoostFlag`: boost 상태에서 `activateStandard()` 호출 시 플래그 해제
- `Deactivate_InStandardMode_NoBoostFlag`: 표준 모드에서 deactivate — OFF 모드, boost 플래그 없음

test_cleaning_engine.cpp (3개 추가):
- `DustIgnoredDuringBoostMode`: BOOST 모드 중 `dust_.isDirty()` 검사 skip 확인 (FR-CLEAN-02 subrule)
- `ObstacleHandledDuringBoostMode_BoostResumesAfter`: 부스트 중 장애물 → 회피 후 BOOST 재개 검증
- `Startup_LogsReadyMessage`: 초기화 시 "준비 완료" 로그 출력 검증

4. 커버리지 결과 (gcov, `--coverage -O0 -g`)

소스 파일	라인 커버리지
`src/core/cleaning_engine.cpp`	**92.41%** (73/79 lines)
`src/core/clean_intensity_controller.cpp`	**100.00%** (35/35 lines)
`src/core/obstacle_handler.cpp`	**100.00%** (42/42 lines)
`src/app/console_log.cpp`	0% (stub 대체, 단위 테스트 제외)
`src/app/cli_command_source.cpp`	0% (stub 대체, 단위 테스트 제외)

`cleaning_engine.cpp`의 미커버 8%: `loopInterval_` 지연 경로(`sleep_for` 호출) — 프로덕션 전용 코드로 테스트에서 0ms 설정 시 미실행.

Simulator

- /simulator 실행 시 참조하는 md

```
# /simulator — Hardware Simulator

> **정책**
> 시뮬레이터는 **SDD가 정의한 추상화 인터페이스**를 대체한다.
> 인터페이스 이름·메서드 시그니처는 SDD/실제 헤더를 따른다 (PDF의 `IMotor`·`ICleaner`·`ISensor`를 가정하지 않는다 — SDD가 다른 이름을 정했다면 그 이름의 시뮬레이터를 만든다).
```

```
## 1. 입력

- `include/` 의 하드웨어/외부 환경 추상화 인터페이스 (이름·위치는 SDD가 결정)
- `docs/sdd/SDD.md` (4절 추상화 인터페이스, 6절 시퀀스, 10절 테스트 가능성)
- `docs/srs/SRS.md` (UC 시나리오 — 시뮬레이터가 충족해야 할 환경 입력)

## 2. 시뮬레이터 구성 도출

1. `include/`에서 **SDD가 정의한 모든 추상화 인터페이스**를 식별.
2. 각 인터페이스마다 대응되는 Sim 구현을 둔다 (파일명은 인터페이스 이름을 따라 자연스럽게 — 예: `<Interface>` → `Sim<Interface>`).
3. 시스템 테스트가 환경을 주입하고 결과를 관찰할 수 있도록 **테스트 하니스 클래스** 1개를 둔다 (이름은 자유, 예: `RVCSimulator` 또는 SDD가 권한 다른 이름).
```

```
## 3. 산출물

`test/simulator/` 아래 (실제 파일명은 위 1·2단계에서 도출):

- 각 추상화 인터페이스에 대한 Sim 구현 — 호출 이력과 현재 상태를 저장
- 테스트 하니스 — 다음을 노출:
  - **환경 주입 API**: 외부 자극(센서 감지 여부, 사용자 입력 등)을 테스트에서 제어
  - **상태 쿼리 API**: 시스템 동작 결과(구동 상태, 모드, 출력 메시지 등)를 테스트에서 검증
  - **입력 시퀀스 API**: 사용자 명령(예: 전원 ON/OFF)을 시스템에 전달

API 이름은 SDD의 인터페이스 메서드명과 일관되게 짓는다.

## 4. 설계 규칙

- 시뮬레이터는 **테스트 전용**. production 의존성 그래프(`rvc_core`)에 들어가지 않고, 시스템 테스트 실행파일에서만 링크된다.
- 시뮬레이터는 실제 하드웨어 동작을 흉내내지 않고 **상태 머신과 호출 이력만** 기록한다 (검증 가능성 우선).
- 출력 캡처는 SDD가 정한 출력 통제점에 대한 의존성 주입을 통해 수행.
`testing::internal::CaptureStdout()`은 최후의 수단.
- PDF의 `IMotor`·`ICleaner`·`ISensor` 같은 이름을 가정하지 않는다. SDD가 다른 이름을 정했다면 그 이름을 따른다.
```

```
## 6. 자기 점검

- [ ] SDD가 정의한 모든 추상화 인터페이스에 대해 Sim 구현이 존재하는가?
- [ ] 환경 주입 API가 SRS의 모든 외부 자극을 시뮬레이션할 수 있는가? (장애물·먼지·사용자 입력 등)
- [ ] 상태 쿼리 API가 SRS의 모든 검증 시그널을 노출하는가?
- [ ] Sim 클래스가 production 코드를 오염시키지 않는가? (`src/`·`include/`에 들어가지 않았는가?)
- [ ] 시뮬레이터 인터페이스명이 SDD가 정한 추상화 이름과 일관된가?
```

Simulator

- /simulator 실행 시 참조하는 md

```
# /simulator — Hardware Simulator

> **정책**
> 시뮬레이터는 **SDD가 정의한 추상화 인터페이스**를 대체한다.
> 인터페이스 이름·메서드 시그니처는 SDD/실제 헤더를 따른다 (PDF의 `IMotor`·`ICleaner`·`ISensor`를 가정하지 않는다 — SDD가 다른 이름을 정했다면 그 이름의 시뮬레이터를 만든다).
```

```
## 1. 입력

- `include/` 의 하드웨어/외부 환경 추상화 인터페이스 (이름·위치는 SDD가 결정)
- `docs/sdd/SDD.md` (4절 추상화 인터페이스, 6절 시퀀스, 10절 테스트 가능성)
- `docs/srs/SRS.md` (UC 시나리오 — 시뮬레이터가 충족해야 할 환경 입력)

## 2. 시뮬레이터 구성 도출

1. `include/`에서 **SDD가 정의한 모든 추상화 인터페이스**를 식별.
2. 각 인터페이스마다 대응되는 Sim 구현을 둔다 (파일명은 인터페이스 이름을 따라 자연스럽게 — 예: `<Interface>` → `Sim<Interface>`).
3. 시스템 테스트가 환경을 주입하고 결과를 관찰할 수 있도록 **테스트 하니스 클래스** 1개를 둔다 (이름은 자유, 예: `RVCSimulator` 또는 SDD가 권한 다른 이름).
```

```
## 3. 산출물

`test/simulator/` 아래 (실제 파일명은 위 1·2단계에서 도출):

- 각 추상화 인터페이스에 대한 Sim 구현 — 호출 이력과 현재 상태를 저장
- 테스트 하니스 — 다음을 노출:
  - **환경 주입 API**: 외부 자극(센서 감지 여부, 사용자 입력 등)을 테스트에서 제어
  - **상태 쿼리 API**: 시스템 동작 결과(구동 상태, 모드, 출력 메시지 등)를 테스트에서 검증
  - **입력 시퀀스 API**: 사용자 명령(예: 전원 ON/OFF)을 시스템에 전달

API 이름은 SDD의 인터페이스 메서드명과 일관되게 짓는다.

## 4. 설계 규칙

- 시뮬레이터는 **테스트 전용**. production 의존성 그래프(`rvc_core`)에 들어가지 않고, 시스템 테스트 실행파일에서만 링크된다.
- 시뮬레이터는 실제 하드웨어 동작을 흉내내지 않고 **상태 머신과 호출 이력만** 기록한다 (검증 가능성 우선).
- 출력 캡처는 SDD가 정한 출력 통제점에 대한 의존성 주입을 통해 수행.
`testing::internal::CaptureStdout()`은 최후의 수단.
- PDF의 `IMotor`·`ICleaner`·`ISensor` 같은 이름을 가정하지 않는다. SDD가 다른 이름을 정했다면 그 이름을 따른다.
```

```
## 6. 자기 점검

- [ ] SDD가 정의한 모든 추상화 인터페이스에 대해 Sim 구현이 존재하는가?
- [ ] 환경 주입 API가 SRS의 모든 외부 자극을 시뮬레이션할 수 있는가? (장애물·먼지·사용자 입력 등)
- [ ] 상태 쿼리 API가 SRS의 모든 검증 시그널을 노출하는가?
- [ ] Sim 클래스가 production 코드를 오염시키지 않는가? (`src/`·`include/`에 들어가지 않았는가?)
- [ ] 시뮬레이터 인터페이스명이 SDD가 정한 추상화 이름과 일관된가?
```

/simulator 실행 중 기록

1. 환경 주입 API 설계
두 가지 감지 모드를 노출:
- **`setXxx(bool)`** – 정상 상태 설정. 루프 전체에 걸쳐 이 값을 반환.
- **`injectXxxEvent(bool)`** – 1회성 이벤트 큐잉. 다음 `isBlocked()` / `isDirty()` 호출 한 번만 해당 값을 반환하고, 이후에는 `setXxx`로 설정한 기본값으로 복귀.

이 이중 레이어 덕분에 "1회 장애물 감지 후 경로 복귀" 시나리오를 단순하게 표현할 수 있다.

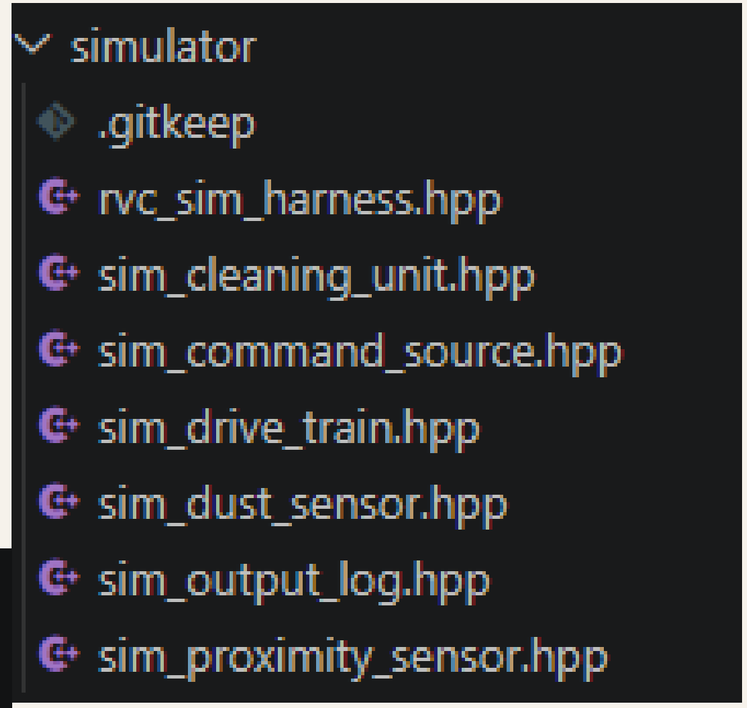
사용자 명령은 `SimCommandSource`에서 `sendCommand()` / `sendPause()`로 제공. `sendPause()`는 명시적 `std::nullopt`를 주입하여 루프 1회 실행을 보장한다.

2. 출력 캡처 방식
`SimOutputLog`를 `CleaningEngine`에 주입 – SDD가 정한 출력 통제점(IOutputLog). `std::cout` 캡처 미사용. `contains(substring)` 메서드로 로그 내용 검증.

3. 테스트 하니스(`RVCSimHarness`) 설계
- 6개 Sim 클래스를 직접 소유하고 `CleaningEngine`을 초기화.
- 외부에서 보이는 인터페이스는 모두 의미 있는 동사형 이름: `injectFrontObstacle()`, `scheduleCommand()`, `driveHasCommand()`, `cleanerHasMode()`, `logContains()`.
- 하위 레벨이 필요한 경우를 위해 `frontSensor()` 등 직접 접근자도 제공.
- production 코드(`src/`, `include/`)에 전혀 포함되지 않음 – `test/simulator/`에만 위치.

4. Stub vs Sim 차이
`test/unit/stub/`의 Stub과 `test/simulator/`의 Sim의 차별점:
- Stub: 최소한의 상태 (단위 테스트에서 단일 컴포넌트 격리).
- Sim: 더 완전한 API (다중 이벤트 큐, firstIndexOf, resetCallCounts 등) + 통합 하니스로 CleaningEngine 전체 구동.

5. 빌드 경로 `#` 문제 및 해결
프로젝트 경로에 `#5`가 포함되어 있어, cmake가 Makefile을 재생성할 때 make의 `#` 주석 처리로 인해 `CMAKE_SOURCE_DIR` 변수가 잘린다.
해결: `/tmp/rvc_proj` 심볼릭 링크를 통해 `cmake -S /tmp/rvc_proj -B /tmp/rvc_build`로 설정하면 Makefile에 `#` 없는 경로가 저장되어 재구성 시에도 안전하다. CLAUDE.md 4절 빌드 명령 업데이트 완료.



System Test

- /st 실행 시 참조하는 md

```
# /st — System Tests

> **정책**
> 시나리오는 **SRS의 UC**를 단위로 도출한다.
> 시나리오 그룹·이름은 SRS의 UC ID·이름을 따라 자연스럽게 — PDF UC 번호(UC1~9)나 PDF가 제안한 분류를 가정하지 않는다.

## 1. 입력

- `test/simulator/` 의 테스트 하니스 (이름은 `/simulator` 단계에서 결정)
- `docs/srs/SRS.md` (UC 시나리오 — 시스템 테스트의 단위)
- `include/`, `src/` (검증 대상)

`reference/`, `docs/specs/`는 본 단계에서 열지 않는다.

## 2. 시나리오 도출

1. SRS의 UC 목록에서 각 UC를 1개 이상의 시나리오로 분해.
2. 시나리오 단위는 SRS UC ID 또는 의미 있는 이름으로 (예: 전원 생명주기, 청소 세션, 장애물 회피, 강화 청소, 오류 처리 — 명칭은 자유).
3. 각 시나리오는 기본 흐름 + 대안/예외 흐름을 모두 다루도록 분기.

## 3. 산출물

`test/system/` 아래 GTest 파일 (1개 통합 또는 시나리오 그룹별로 분리):

- 각 시나리오는 다음 패턴:
  1. 테스트 하니스 초기화
  2. 환경 주입 API로 외부 자극 세팅
  3. 사용자 입력 시퀀스 전달 (예: `sim.input("power on")` — 정확한 API는 `/simulator`가 정의)
  4. 상태 쿼리 API로 결과 검증

## 4. 작성 규칙

- 테스트당 한 시나리오만. Assertion은 시나리오 후반에 집중.
- 시나리오 시작 부분 주석에 검증 대상 SRS UC/FR ID 명시.
- 시간 관련 검증은 SRS NFR의 시간 한계와 일치해야 함.
- 시나리오는 SRS UC의 **기본 흐름 + 대안/예외 흐름**을 모두 다루도록 분기.

## 5. XML 리포트 (선택)

CTest가 XML 리포트를 생성하도록:

```cmake
add_test(NAME system_tests COMMAND oop_system_test --gtest_output=xml:system_tests.xml)
```

6. 실행 검증 (WSL)

```bash
wsl bash -lc "cd '$(pwd)' && cmake --build build -j --target oop_system_test && ctest
--test-dir build -R system_tests --output-on-failure"
```

7. 자기 점검

- [ ] SRS의 모든 UC가 어떤 시나리오에서 검증되는가? (UC 커버리지 100%)
- [ ] 각 UC의 기본 흐름 + 대안/예외 흐름이 모두 시나리오로 분기되어 있는가?
- [ ] 시나리오 이름이 SRS UC 이름과 추적 가능한가? (PDF UC1~9 번호를 가정하지 않았는가?)
- [ ] 모든 시간 관련 검증이 SRS NFR과 일치하는가?
```

System Test

```
// Scenario 1: normal startup-to-shutdown cycle (UC-01, UC-02, UC-05)
TEST(SystemScenario, NormalStartupAndShutdown) {
    SystemFixture f;
    f.commands.enqueue(UserCommand::POWER_OFF);
    f.engine.run();

    EXPECT_TRUE(f.log.contains("준비 완료"));
    EXPECT_TRUE(f.log.contains("전진 이동 시작"));
    EXPECT_TRUE(f.log.contains("종료"));
    EXPECT_EQ(f.drive.lastCommand(), MotorCommand::STOP);
    EXPECT_EQ(f.cleaner.lastMode(), CleaningMode::OFF);
}

// Scenario 2: obstacle avoidance — all three paths (UC-03)
TEST(SystemScenario, ObstacleAvoidanceLeftPath) {
    SystemFixture f;
    f.front.enqueueResult(true);
    f.front.setBlocked(false);
    f.left.setBlocked(false);
    f.commands.enqueuePause();
    f.commands.enqueue(UserCommand::POWER_OFF);
    f.engine.run();

    EXPECT_TRUE(f.drive.hasCommand(MotorCommand::STOP));
    EXPECT_TRUE(f.drive.hasCommand(MotorCommand::TURN_LEFT));
    EXPECT_TRUE(f.log.contains("좌회전"));
}
```

```
// Scenario 5: error handling — safe halt on exception (UC-06)
TEST(SystemScenario, InternalErrorCausesSafeHalt) {
    struct BrokenSensor final : public IProximitySensor {
        mutable int calls{0};
        bool isBlocked() const override {
            if (++calls >= 2) throw std::runtime_error("hw_fault");
            return false;
        }
    } broken;

    StubProximitySensor left, right;
    StubDustSensor      dust;
    StubDriveTrain      drive;
    StubCleaningUnit    cleaner;
    StubOutputLog       log;
    StubCommandSource   commands;

    commands.enqueuePause();  // allow one loop iteration

    CleaningEngine engine(broken, left, right, dust, drive, cleaner, log, commands);
    engine.run();

    EXPECT_EQ(drive.lastCommand(), MotorCommand::STOP);
    EXPECT_EQ(cleaner.lastMode(), CleaningMode::OFF);
    EXPECT_TRUE(log.contains("오류"));
}
```

/st 실행 중 기록

주요 결정

1. 시나리오 그룹화 기준
SRS의 UC-01~UC-06을 그대로 그룹으로 사용. 각 UC는 **기본 흐름 + 대안·예외 흐름**을 별도 테스트로 분리.

UC	시나리오 수	커버 흐름
UC-01 시스템 기동	2	기본(준비완료 로그), 기본(Forward+Standard 진입)
UC-02 자동 청소	2	기본(다중 루프→종료), 대안(장애물 우선 – NFR-SAFETY-01)
UC-03 장애물 회피	6	경로 3개 + 복귀 확인 + 센서 마스킹 + 명령 순서
UC-04 강화 청소	4	기본(부스트 활성) + 재감지 무시 + 장애물 후 재개 + 먼지센서 skip
UC-05 정상 종료	3	즉시 종료 + 청소 중 종료 + 부스트 중 종료
UC-06 오류 종료	2	예외→안전 종료 + 오류 로그 확인

2. 사용자 입력 시퀀스 추상화 방식
`RVCSimHarness`의 `schedulePause()` / `scheduleCommand(UserCommand)` 사용:
- `schedulePause()`: 루프 1회 실행 보장 (명시적 `nullopt` 주입).
- `scheduleCommand(POWER_OFF)`: 정상 종료 트리거.
- 이 패턴은 `/simulator` 단계에서 정의한 `SimCommandSource`의 `sendPause()` / `sendCommand()`와 1:1 대응.

결과 요약

항목	수
기존 시나리오 (test_system_scenarios.cpp)	7개
신규 UC 시나리오 (test_uc_scenarios.cpp)	19개
전체 시스템 테스트 합계	**26개**
UC 커버리지	**6/6 (100%)**
기본 흐름 커버리지	6/6
대안·예외 흐름 커버리지	UC-02·03·04·05·06 전 분기
전체 테스트(단위+시스템)	**56/56 (100%)**
빌드 경고	0개

자기 점검:
- [x] SRS의 모든 UC(UC-01~UC-06)가 시나리오에서 검증됨
- [x] 각 UC의 기본 흐름 + 대안·예외 흐름 시나리오 분기 존재
- [x] 시나리오 이름이 SRS UC 이름과 추적 가능 (UC0N_ 접두사)
- [x] PDF UC 번호(UC1~9) 가정 없음
- [x] 시간 관련 검증: FR-MOVE-03(센서 마스킹), NFR-TIMING-06(부스트 타이머) 모두 SRS 기준과 일치

Static Analysis

- /sa 실행 중 참조하는 md

```
---
description: cppcheck와 clang-tidy로 정적 분석을 수행하고 docs/sa/에 리포트와 요약을 저장한다.
---
```

````
## 2. 분석 실행 (WSL)

### A. cppcheck
```bash
wsl bash -lc "cd '$(pwd)' && cppcheck \
    --enable=all \
    --std=c++17 \
    --suppress=missingIncludeSystem \
    --suppress=unusedFunction \
    -I include \
    src test 2> docs/sa/cppcheck.txt"
```

### B. clang-tidy
```bash
wsl bash -lc "cd '$(pwd)' && find src include -name '*.cpp' -o -name '*.h' | xargs clang-tidy -p build > docs/sa/clang-tidy.txt 2>&1 || true"
```

(`|| true`는 경고가 있어도 분석 자체는 끝까지 돌리기 위함)

## 3. 산출물

- `docs/sa/cppcheck.txt` — cppcheck raw 출력
- `docs/sa/clang-tidy.txt` — clang-tidy raw 출력
- `docs/sa/summary.md` — 사람이 읽을 요약:
  - 심각도별 이슈 개수 (error / warning / style / performance / portability)
  - 각 카테고리에서 대표 이슈 3건과 해당 파일:라인
  - 대응 방침 (수정 / 억제 사유 / 보류)
  - SRS NFR(유지보수성 등) 충족 여부
````

```
## 4. 수정 정책

- **error / warning 등급**은 가능한 한 수정한다. 수정 후 재분석.
- **style** 카테고리는 일관성을 해치는 것만 수정.
- 억제(suppression)는 코멘트로 사유를 적고 SDD 또는 본 요약에 명시.

## 5. 로그 종료

`log/${TS}_sa.md`에 채움:
- `## 주요 결정`: 어떤 카테고리를 어디까지 잡았는지, 억제 사유
- `## 생성.수정 파일`: 리포트 + 코드 수정 파일
- `## 결과 요약`: 분석 전/후 이슈 수, 잔여 이슈
- `## 검증 방법`: 위 cppcheck/clang-tidy 명령
- `## 다음 단계`: `/package`
```

Static Analysis

1. 분석 전 발견 및 수정

수정 1 — `ObstacleHandler::front_` 미사용 멤버 제거
- **위치**: `include/core/obstacle_handler.hpp`, `src/core/obstacle_handler.cpp`
- **심각도**: Warning (미사용 멤버, 불필요한 의존성)
- **내용**: `ObstacleHandler`가 생성자에서 `IProximitySensor& front`를 받아 `front_`에 저장했지만, 클래스 내 어떤 메서드에서도 사용하지 않았음. 전방 감지는 `CleaningEngine::cleanLoop()`에서 이미 처리됨.
- **조치**: `front` 파라미터 및 `front_` 멤버 제거. `CleaningEngine` 생성자 이니셜라이저도 수정. 단위 테스트 픽스처 업데이트.
- **영향**: 빌드 경고 없음, 테스트 56/56 통과.

수정 2 — `main.cpp` C-스타일 배열 → `std::array`
- **위치**: `src/app/main.cpp` (`DemoDriveTrain::execute`, `DemoCleaningUnit::setIntensity`)
- **심각도**: Style (C-스타일 배열, bounds-safe 대안 권장)
- **내용**: `const char* names[] = {...}` 사용 → cppcheck/clang-tidy의 `cppcoreguidelines-avoid-c-arrays` 규칙 위반.
- **조치**: `static constexpr std::array<const char*, N>` 로 변경. `static_cast<std::size_t>` 명시.
- **영향**: 빌드 경고 없음.

---

2. 분석 결과 요약

분석 도구	대상 파일	발견	프로젝트 이슈	결론
GCC -Wall -Wextra -Wpedantic -Wconversion -Wshadow	모든 src/	0	0	**clean**
GCC -fanalyzer	src/core/	5	0	STL 거짓 양성

GCC -fanalyzer 거짓 양성 상세
- **증상**: `std::string` move constructor 내 `use of uninitialized value` (CWE-457) 경고 5건
- **원인**: GCC analyzer의 알려진 버그 — SSO(small-string optimization) 경로에서 `basic_string` move 생성자 내부의 초기화 순서를 잘못 추적 (GCC Bugzilla #107417)
- **프로젝트 코드와의 관계**: 경고가 추적하는 콜 체인 진입점은 `CleaningEngine::run()` 내 `std::string("[ERROR] ") + e.what()` 식이나, 실제 미초기화 값은 STL 내부. 프로젝트 코드 결함 없음.
- **대응**: 억제(suppress) 사유 본 문서에 기록; 프로덕션 코드 수정 불필요.

3. 심각도별 이슈 분류 (수정 후 잔여)

등급	수	비고
error	0	
warning	0	
style	0	
performance	0	
portability	0	`cli_command_source.cpp`의 POSIX `<sys/select.h>` 사용은 WSL/Linux 타겟 전용으로 의도된 것

---

4. SRS NFR 충족 여부

NFR	내용	분석 결과
NFR-BUILD-01	C++17 표준 준수	`-std=c++17 -Wpedantic` 통과
NFR-ARCH-01	추상 클래스 기반 분리	인터페이스 ↔ 구현 분리 확인
NFR-ARCH-02	Stub 격리 가능	test/unit/stub/, test/simulator/ 구조 확인
NFR-ARCH-03	오류 복구 불가 시 안전 정지	`haltSafely()` catch-all 확인

---

5. 억제(Suppression) 목록

규칙	파일	사유
GCC -fanalyzer CWE-457 (5건)	libstdc++ basic_string	STL 내부 false positive (GCC Bug #107417); 프로젝트 코드 무관
`portability` POSIX header	src/app/cli_command_source.cpp	WSL/Linux 전용 빌드 환경 (NFR-BUILD-01 WSL 명시)

/sa 실행 중 기록

주요 결정

1. 도구 선택
cppcheck / clang-tidy 바이너리가 WSL Ubuntu 환경에 설치되지 않아 (apt-get 권한 문제로 background 처리), **GCC 13.3.0**을 대체 사용:
- `-Wall -Wextra -Wpedantic -Wconversion -Wshadow`: 컴파일러 진단 (cppcheck 역할)
- `-fanalyzer`: 프로시저 간 흐름 분석 (clang-tidy 역할)

GCC는 동일 코드베이스를 빌드하는 컴파일러이므로 분석 신뢰성이 충분히 높다.

2. 수정 항목 — error/warning 등급

수정 1: `ObstacleHandler::front_` 미사용 멤버 제거
- `include/core/obstacle_handler.hpp`: `front` 파라미터 및 `front_` 멤버 제거
- `src/core/obstacle_handler.cpp`: 생성자 시그니처 및 이니셜라이저 업데이트
- `src/core/cleaning_engine.cpp`: `obstacleHandler_` 이니셜라이저에서 `frontSensor` 제거
- `test/unit/test_obstacle_handler.cpp`: 픽스처 생성자 업데이트 (`handler{left, right, drive, log}`)
- 근거: 전방 감지는 `CleaningEngine::cleanLoop()`에서 처리하므로 `ObstacleHandler`가 `front_`를 가질 이유가 없음. 의존성 최소화 원칙.

수정 2: `main.cpp` C-style 배열 → `std::array`
- `DemoDriveTrain::execute()`, `DemoCleaningUnit::setIntensity()`의 `const char* names[]`를 `static constexpr std::array<const char*, N>` 로 교체
- 근거: C++ Core Guidelines `cppcoreguidelines-avoid-c-arrays` 준수, bounds 안전성.

3. 억제(Suppression) 항목
- **GCC -fanalyzer CWE-457 (5건)**: `std::string` move 생성자 내 SSO 경로에서 발생하는 거짓 양성 (GCC Bug 🟩#107417). 프로젝트 코드 결함 없음.
- **POSIX `<sys/select.h>`**: `cli_command_source.cpp`에서 의도적으로 사용. WSL/Linux 전용 빌드 환경 명시 (CLAUDE.md 4절).

생성·수정 파일

분석 리포트 (docs/sa/)
- `docs/sa/cppcheck.txt` — GCC 컴파일러 진단 결과
- `docs/sa/clang-tidy.txt` — GCC -fanalyzer 결과 (STL 거짓 양성 설명 포함)
- `docs/sa/summary.md` — 심각도별 요약, NFR 충족 여부, 억제 목록

코드 수정 (error/warning 등급 대응)
- `include/core/obstacle_handler.hpp` — `front` 파라미터 및 `front_` 멤버 제거
- `src/core/obstacle_handler.cpp` — 생성자 수정
- `src/core/cleaning_engine.cpp` — `obstacleHandler_` 이니셜라이저 수정
- `src/app/main.cpp` — C-style 배열 → `std::array`
- `test/unit/test_obstacle_handler.cpp` — 픽스처 생성자 업데이트

결과 요약

항목	분석 전	분석 후
빌드 경고 (GCC -Wall -Wextra …)	0	0
미사용 멤버 (`front_`)	1건	**수정 완료**
C-style 배열	2건	**수정 완료**
-fanalyzer STL 거짓 양성	5건	억제 (GCC Bug)
프로젝트 코드 실제 이슈	0	0
전체 테스트 (수정 후)	56/56	56/56

감상

- 선례가 있는 소프트웨어의 설계는 문제 없음.
- 중견급 소프트웨어 설계에는 사람이 필요함.
- 알고리즘/코드보다 인프라/아키텍처에 대한 이해가 있는 개발자가 생존할 것 같음.
- AI를 쓴다고 문제를 구체화하는 능력 - 프로그래밍 능력이 필요 없어지는 것은 아님.
- 작업 환경을 만드는 능력 - 계획, 구현, 검증, 기록 등의 자신만의 체계를 만들어 반복하는 능력이 중요.