객체지향개발방법론

1팀 5주차 발표

VIBE CODING

202211327 윤승모
202211261 김강민
202212353 문서인
202011362 정상현

CONTENTS

Part 1: SRS & SDD

Part 2 : Program(Code)

Part 3 : UT & ST

Part 4 : SA & Simulator

Part 0

: Requirements

[AI TOOL]

- Codex 사용
- 요금제는 개인 사용중인 요금제 기준으로

[Prompt]

- 별도의 codex를 사용해 prompt 정제
- Vibe 폴더 속에 작업물들 기반으로 작업하도록 지시

[Source]

- 제출했던 pdf 최신 version 사용
- 작성했던 코드들 대략적으로 scan

Part1 SRS

: Software Requirements Specification

System Scope and Boundary

[Power control]

`PowerButton` 입력에
따른 on/off 전환

[Mode control]

`StandbyMode`, `NormalMode`,
`BoostMode`, `LowBatteryMode` 전환

[Movement control]

Forward, Left, Right,
Backward, Stop 동작 명령

Part1 SRS

: Software Requirements Specification

System Scope and Boundary

[Obstacle handling]

front/left/right obstacle
상태에 따른 회피 방향 결정

[Dust handling]

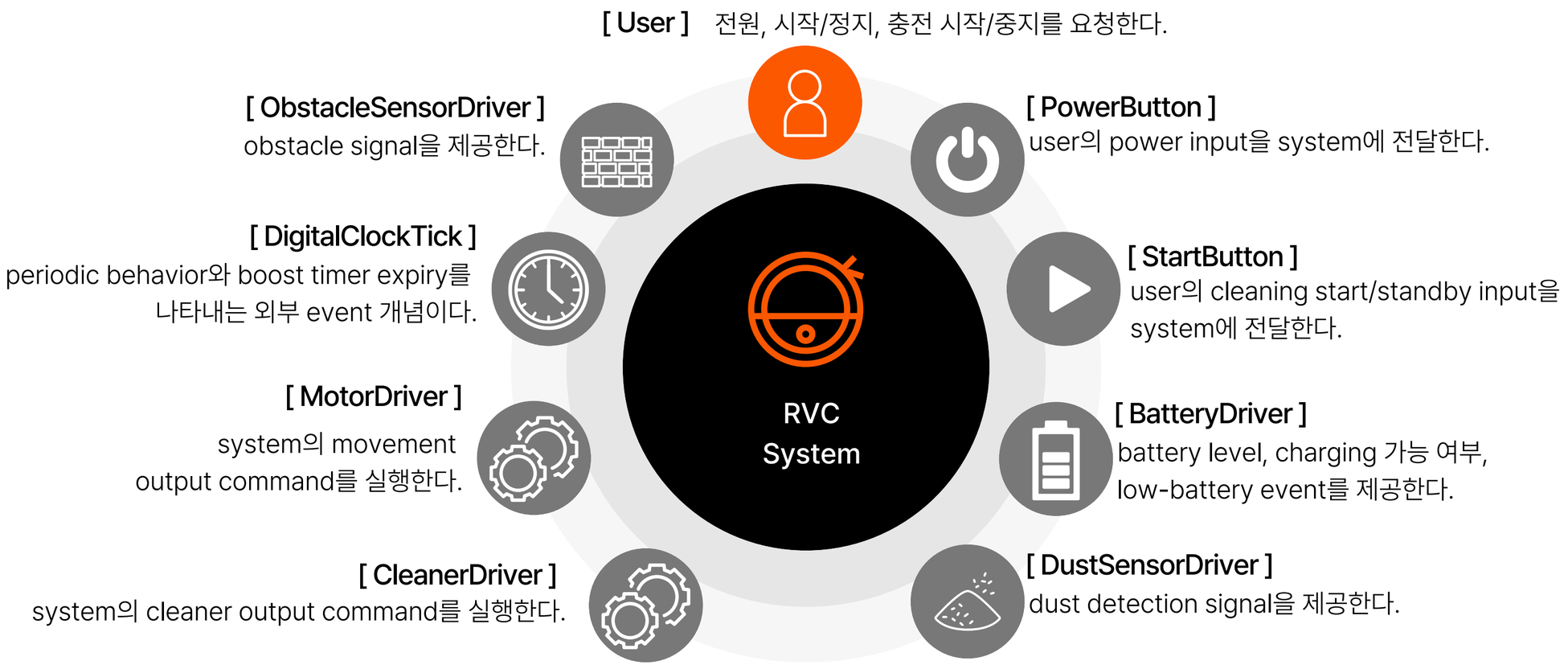
dust signal에 따른 boost mode 전환 및
timer expiry 처리

[Battery handling]

charging, stop charging,
low battery event, recovery 처리

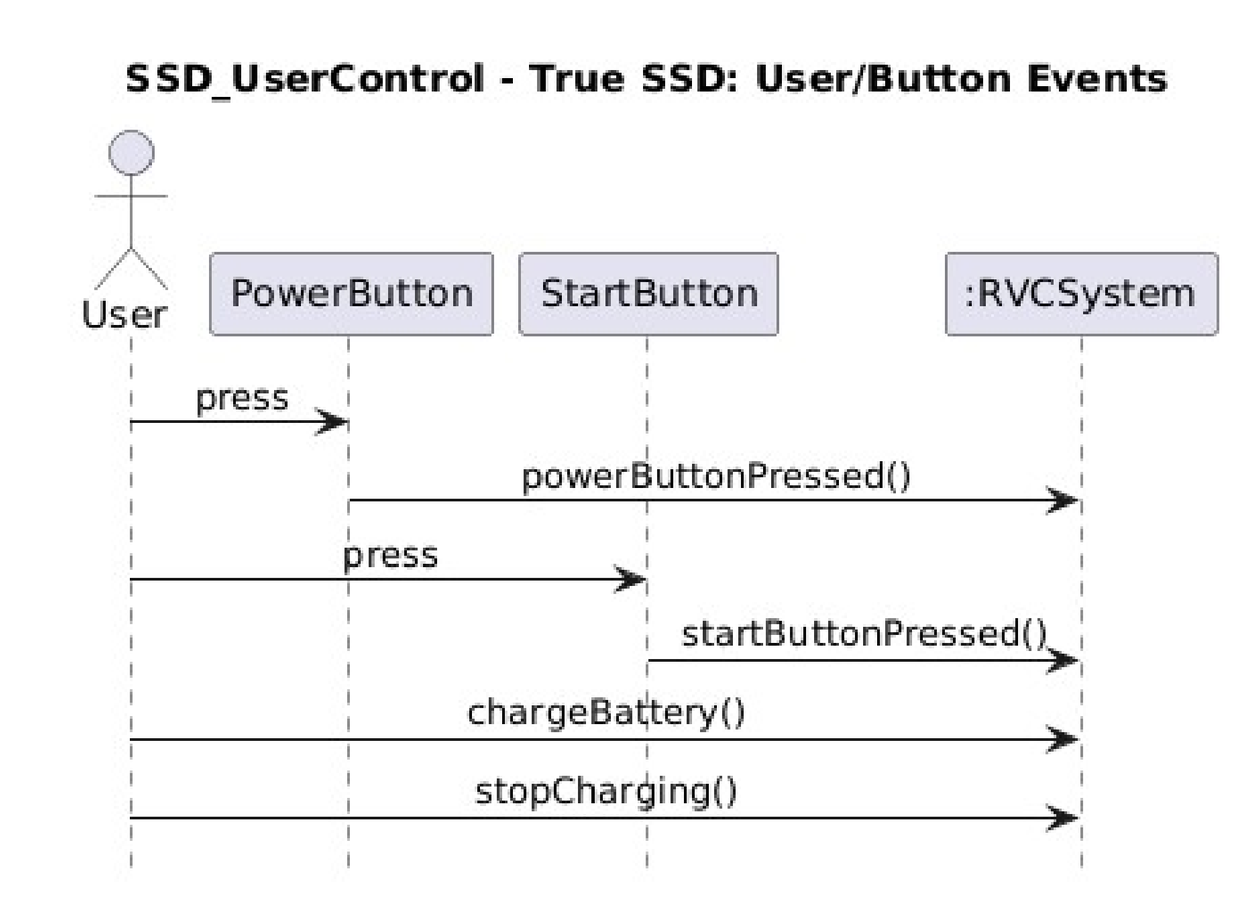
Part1 SRS

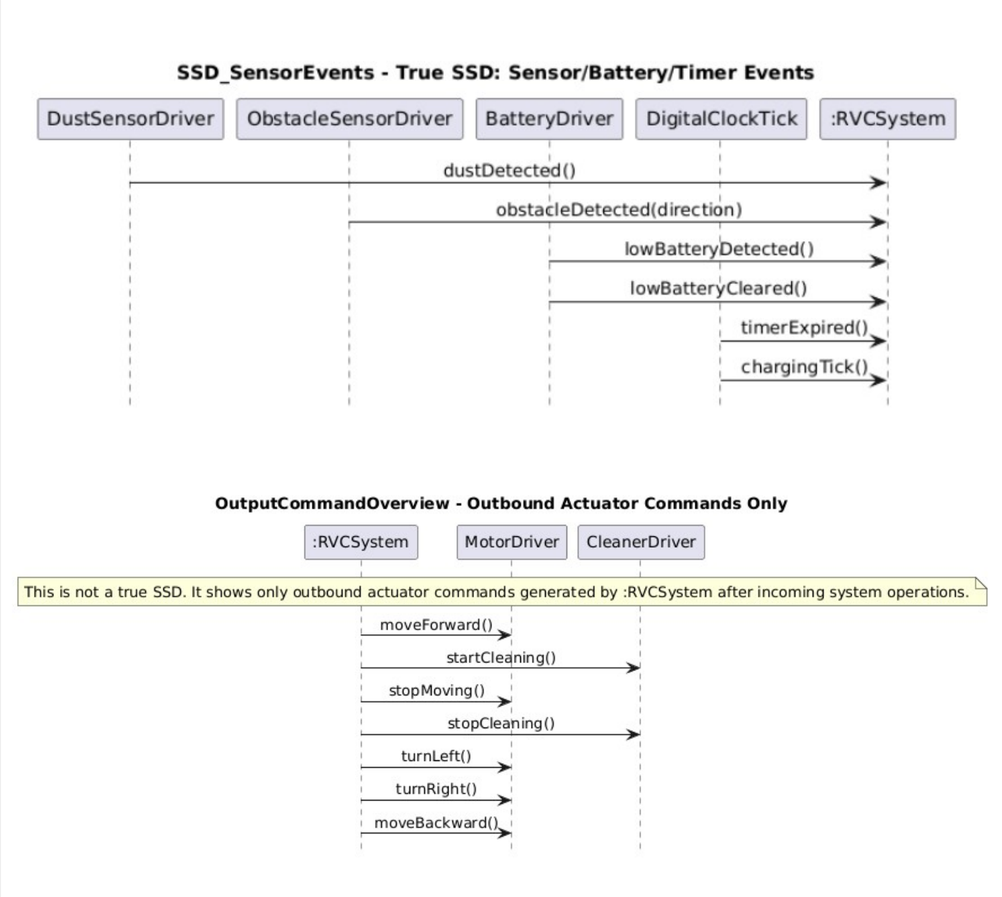
: System Boundary & Actors



Part1 SRS

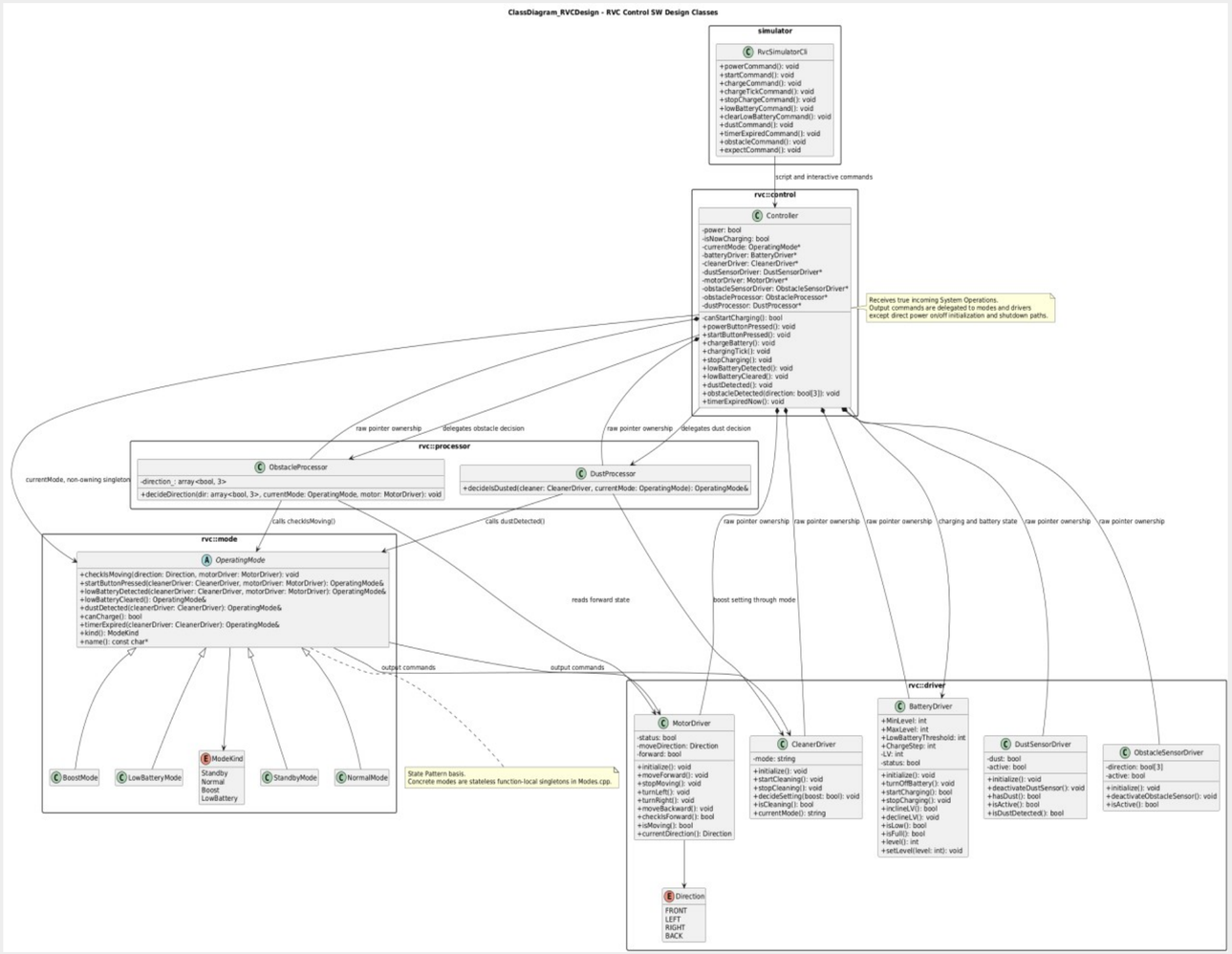
: System Sequence Diagrams





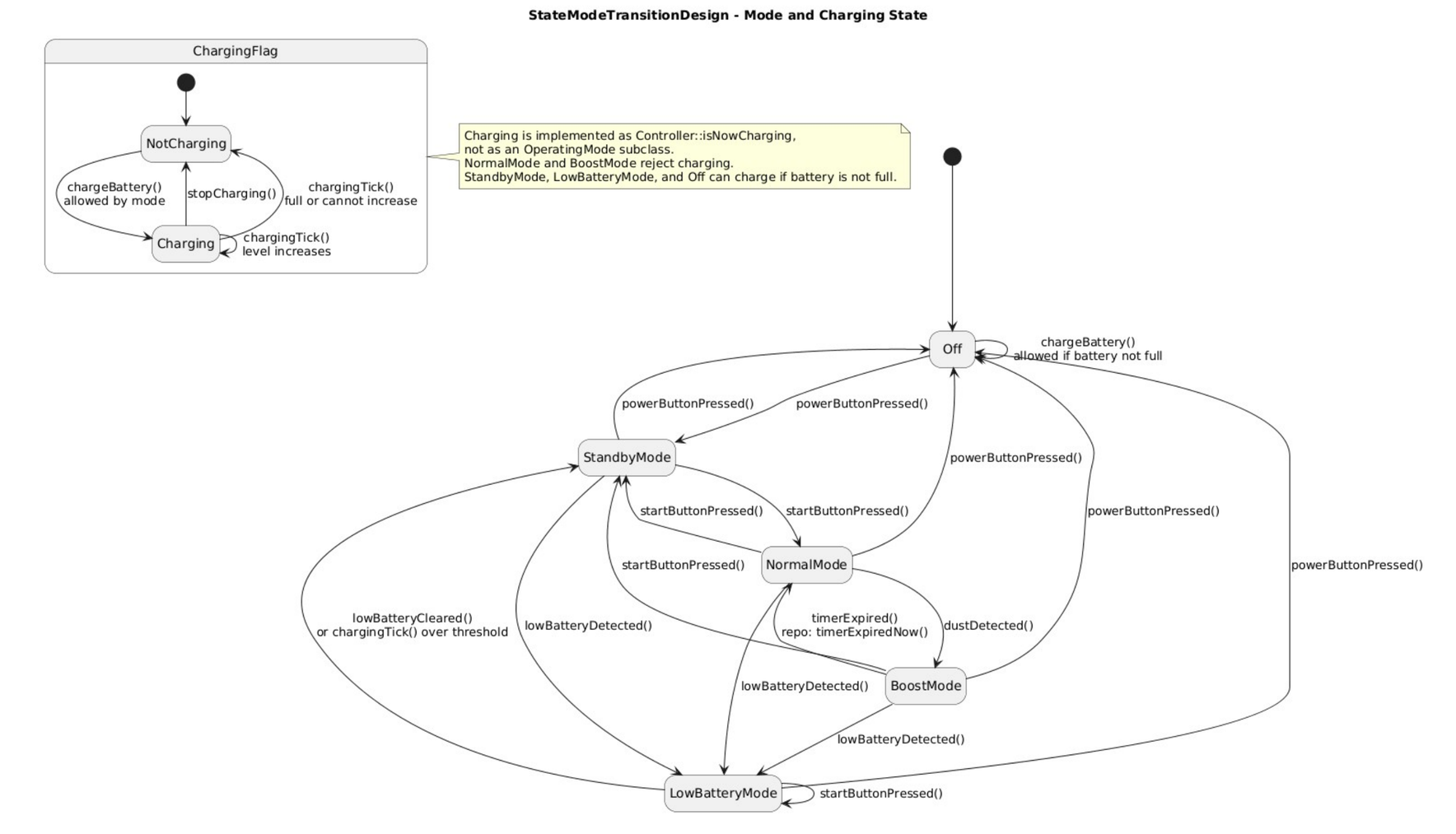
Part1 SDD

: Class Diagram



Part1 SDD

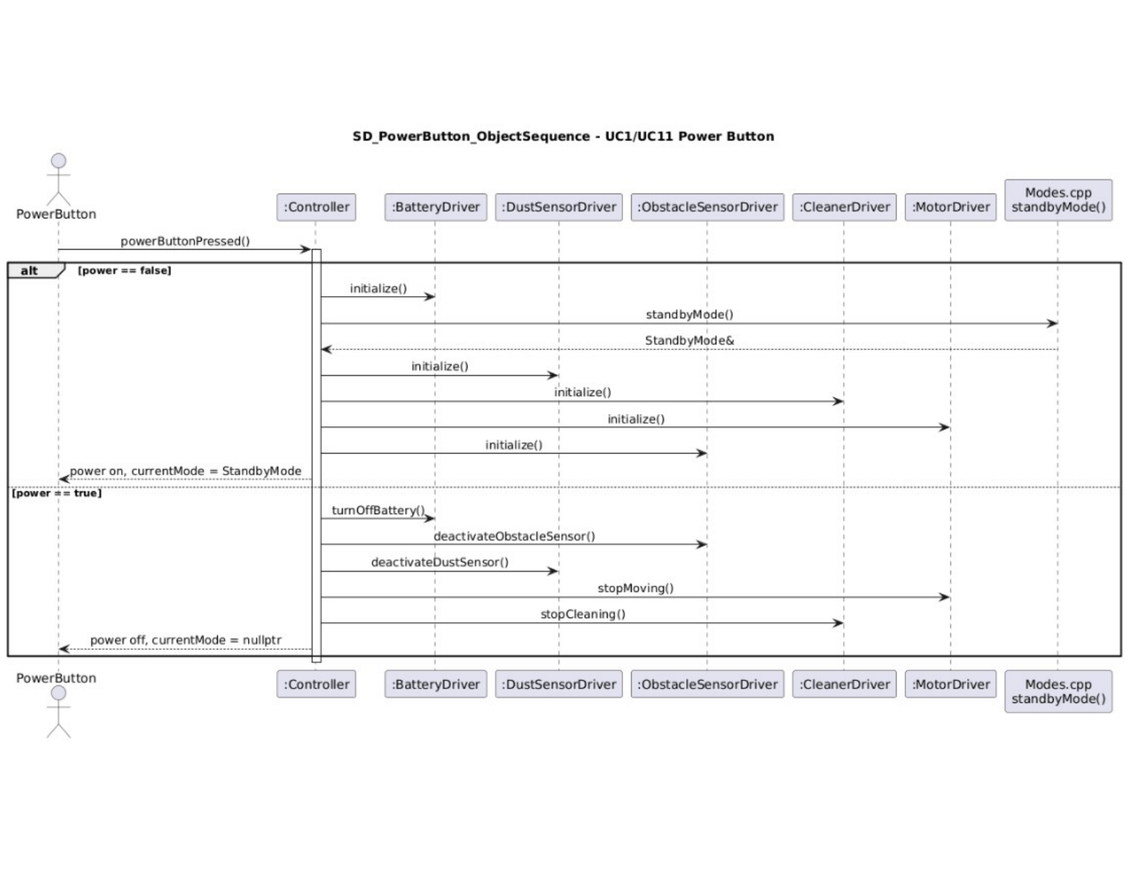
: Mode Transition

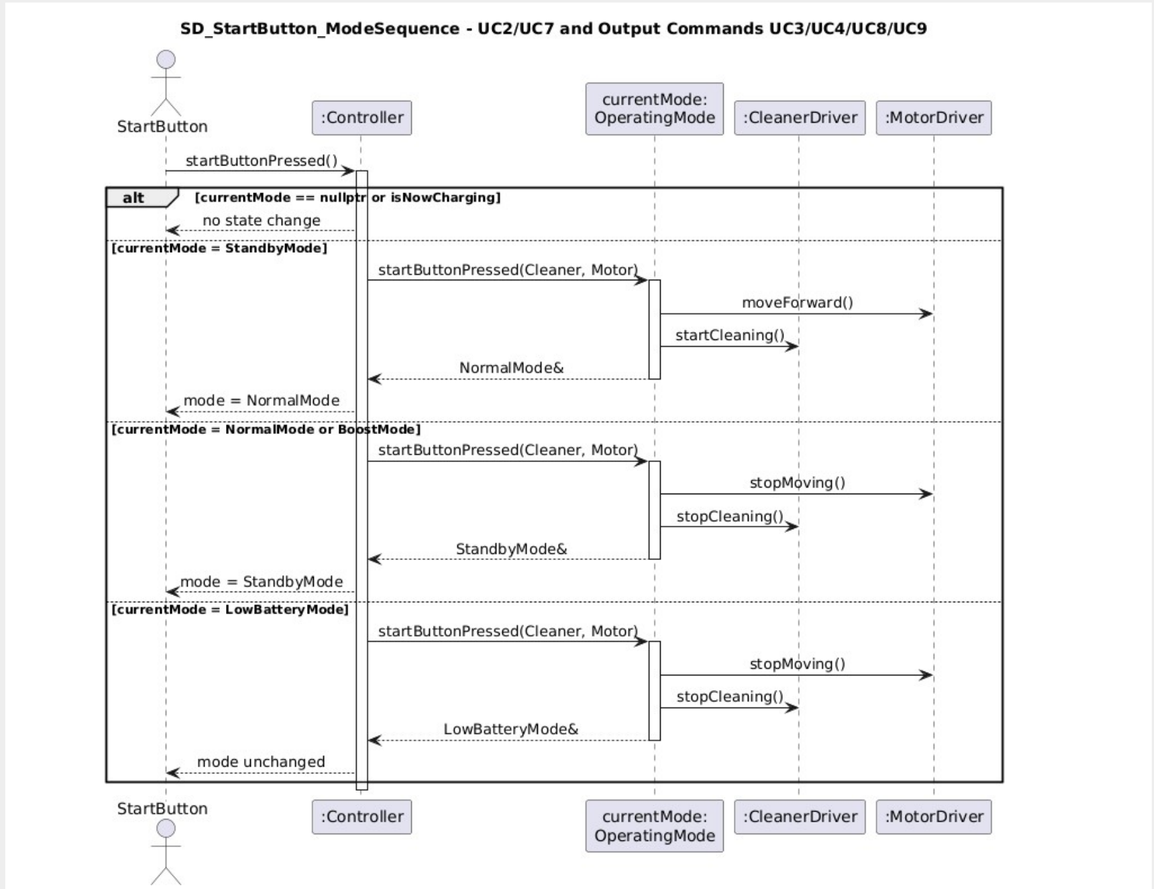


Part1 SDD

: Sequence Diagram

Button Cases

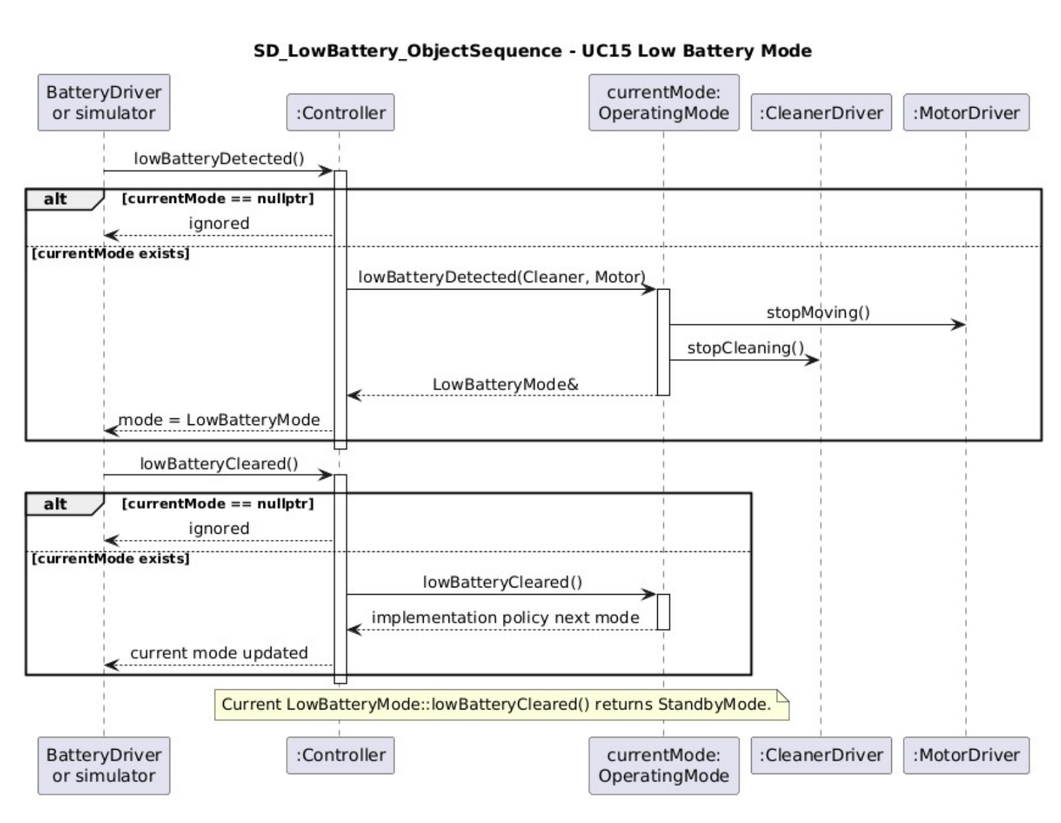


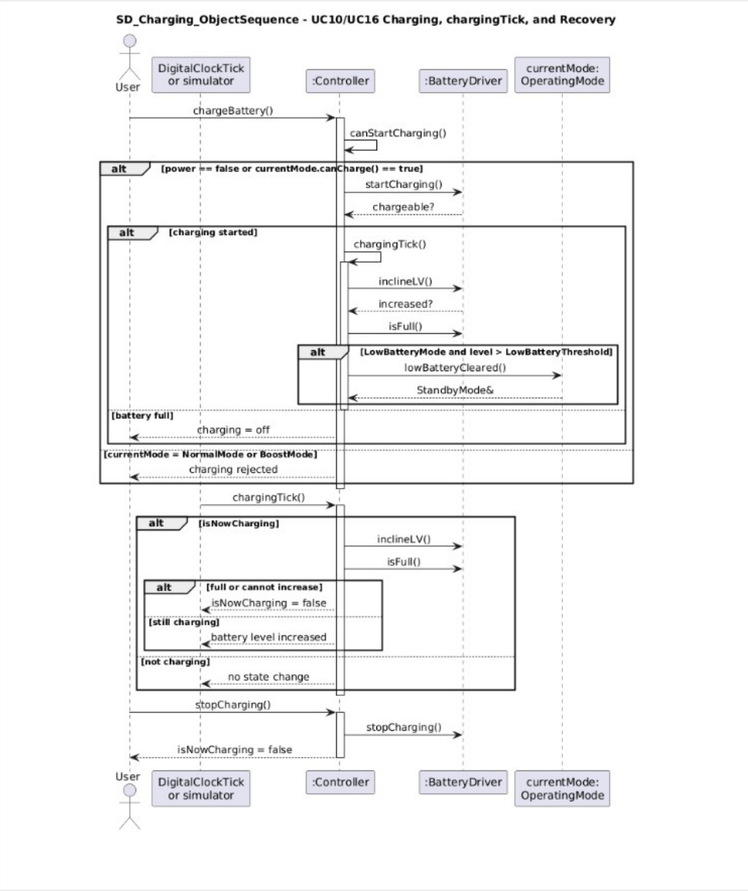


Part1 SDD

: Sequence Diagram

Battery

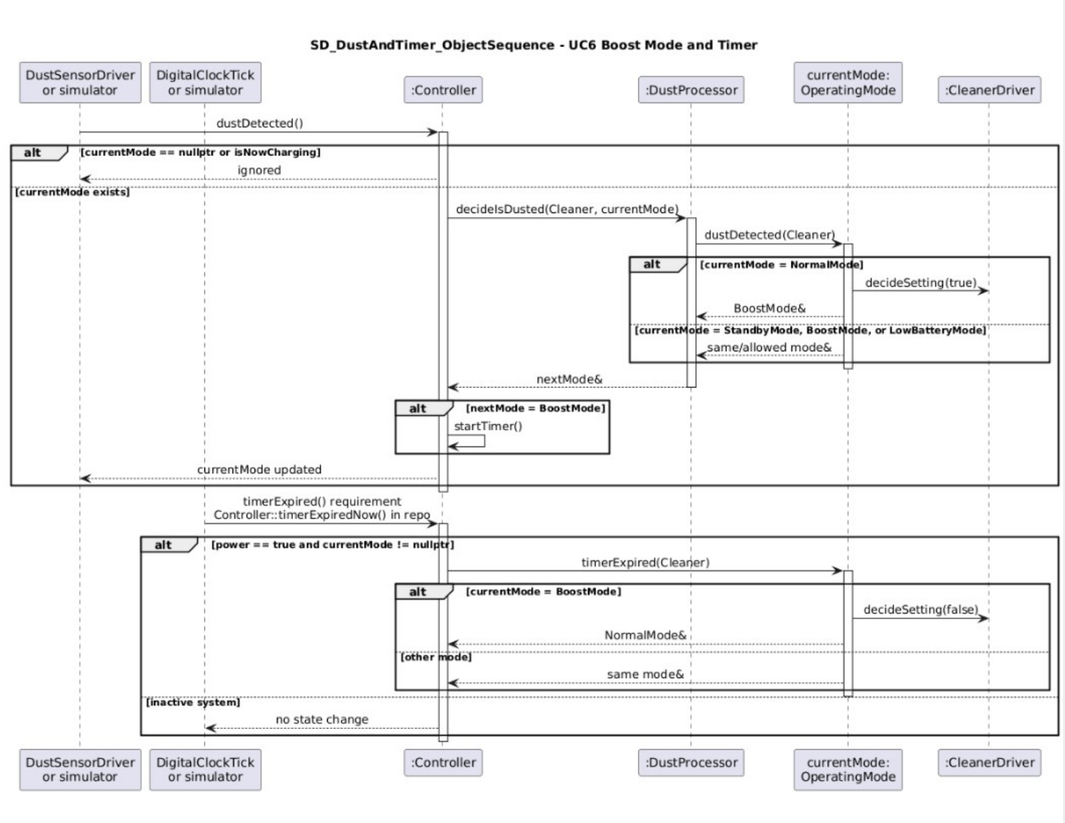




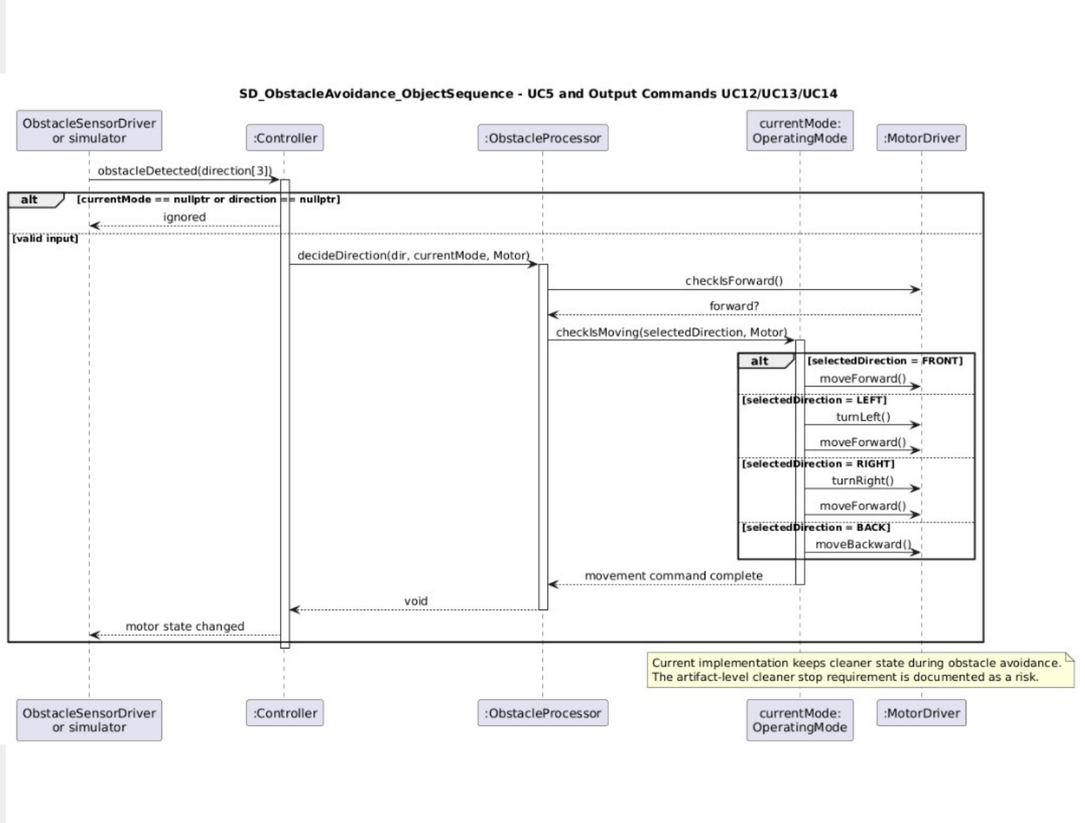
Part1 SDD

: Sequence Diagram

Dust Timer



Obstacle Avoidance



Part2 Program(Code)

: class .h파일(Prompt)

```
Only create NEW files inside vibe/include/rvc/ directory.
Do NOT touch any files outside of vibe/.

Generate header files for all RVC Control SW classes.
Use C++17, include guards.

Constraints:
- Direction: enum class { Forward, Left, Right, Backward }
- OperatingMode: abstract base class, virtual methods:
  startButtonPressed, powerButtonPressed, dustDetected,
  lowBatteryDetected, timerExpired, obstacleDetected
- StandbyMode / NormalMode / BoostMode / LowBatteryMode: concrete subclasses
- Controller: owns all drivers and processors as value members,
  holds OperatingMode* currentMode as raw pointer
  System operations: powerButtonPressed, startButtonPressed,
  chargeBattery, stopCharging, lowBatteryDetected, dustDetected,
  obstacleDetected, timerExpired
- BatteryDriver: bool isCharging, bool isLowBattery
- CleanerDriver: bool isRunning, bool isBoosting
- MotorDriver: bool isRunning, Direction direction
- ObstacleSensorDriver: bool front, left, right
```

Part2 Program(Code)

: Controller(Prompt)

```
Only create NEW files inside vibe/src/ directory.
Do NOT touch any files outside of vibe/.

Implement vibe/src/Controller.cpp based on vibe/include/rvc/Controller.hpp.

Rules:
- No mode-specific logic in Controller; always delegate to currentMode
- powerButtonPressed():
    off → initialize drivers, set currentMode = new StandbyMode, power = true
    on  → stop motor, stop cleaner, delete currentMode, power = false
- startButtonPressed(): delegate to currentMode, update currentMode with return value
- dustDetected(): delegate to currentMode, update currentMode with return value
- lowBatteryDetected(): delete currentMode, set new LowBatteryMode, stop motor and cleaner
- obstacleDetected(): use ObstacleProcessor to pick direction,
  stop cleaner, move accordingly, resume if still in cleaning mode
- chargeBattery(): reject if NormalMode or BoostMode; else batteryDriver.isCharging = true
- timerExpired(): delegate to currentMode, update currentMode with return value
- Mode transition: if returned pointer != currentMode, delete old and assign new
```

Part2 Program(Code)

: Modes(Prompt)

```
Only create NEW files inside vibe/src/ directory.
Do NOT touch any files outside of vibe/.

Implement vibe/src/OperatingMode.cpp based on vibe/include/rvc/OperatingMode.hpp.
Each method returns OperatingMode* (next state) or this if no transition.

StandbyMode:
  startButtonPressed() → return new NormalMode
  others → return this

NormalMode:
  startButtonPressed() → return new StandbyMode
  dustDetected() → return new BoostMode
  lowBatteryDetected() → return new LowBatteryMode
  others → return this

BoostMode:
  startButtonPressed() → return new StandbyMode
  timerExpired() → return new NormalMode
  lowBatteryDetected() → return new LowBatteryMode
  others → return this

LowBatteryMode:
  startButtonPressed() → no-op, return this
  others → return this
```

Part2 Program(Code)

: Processor(Prompt)

```
Only create NEW files inside vibe/src/ directory.
Do NOT touch any files outside of vibe/.

Implement vibe/src/ObstacleProcessor.cpp and vibe/src/DustProcessor.cpp.

ObstacleProcessor:
  Direction decide(bool frontBlocked, bool leftBlocked, bool rightBlocked)
  if !frontBlocked → Forward
  if !leftBlocked  → Left
  if !rightBlocked → Right
  else             → Backward

DustProcessor:
  bool shouldBoost(bool dustDetected, OperatingMode* currentMode)
  return true only if dustDetected == true and currentMode is NormalMode
```

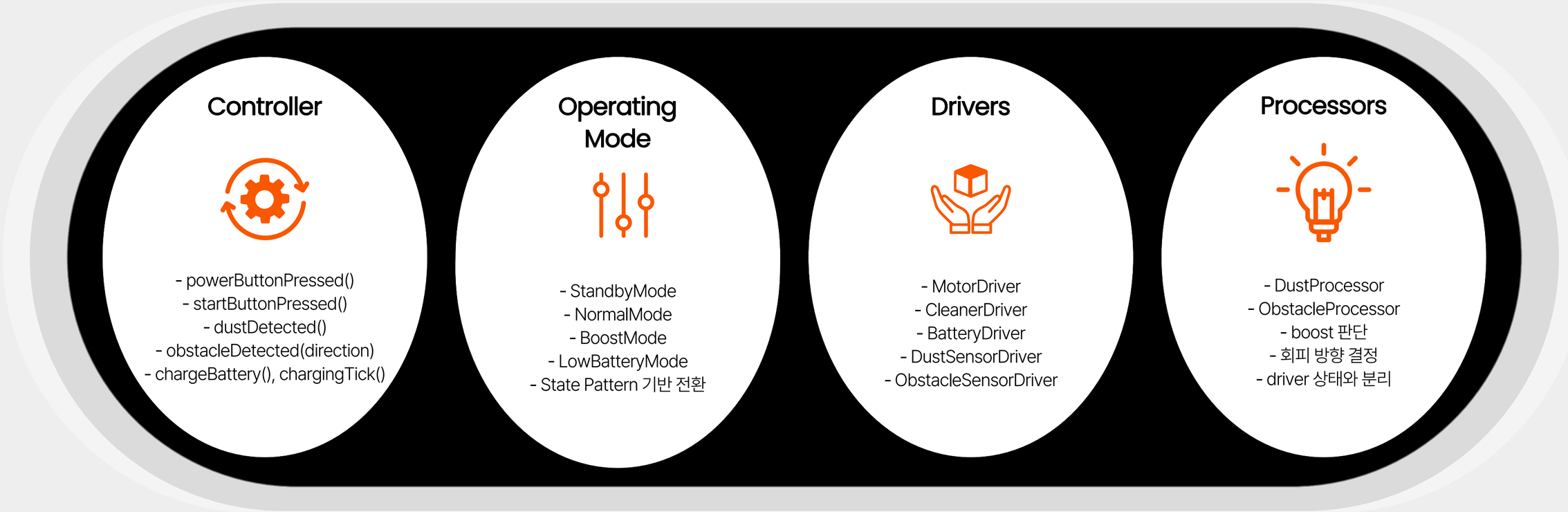
Part2 Program(Code)

: Code Composition

: SRS/SDD에서 정의한 system operation과 mode behavior를 코드, 테스트, simulator가 같은 용어로 재현하도록 정렬했다.

Part2 Program(Code)

: Code Architecture



Controller는 incoming system operation을 받고, mode/driver/processor로 책임을 분산한다.

Part 3

: Unit Test(Prompt)

You are Codex working on the RVC Control SW OOI assignment.

Scope:
- Treat `docs/` and `vibe/` as the current vibe-coding project artifacts.
- First inspect `docs/ai-input/*.md`, `docs/ai-output/srs.md`, `docs/ai-output/sdd.md`, and all current files under `vibe/`.
- Do NOT edit files outside `vibe/`.
- Create unit test files under `vibe/tests/`.
- Use C++17 and Google Test.
- Do not use simulator scripts for unit tests.
- Tests must follow the actual current `vibe/include/rvc/*.hpp` APIs.

Task:
Implement comprehensive Google Test unit tests for the current `vibe/` RVC Control SW implementation.

Testing rules:
- One test suite per system operation or major internal operation.
- Cover normal flow, negative/no-op flow, and edge cases.
- Use fixtures and helper functions.
- Tests must be independent and deterministic.
- Verify mode transition, motor state, cleaner state, battery state, sensor state, charging state, and direction where relevant.
- Respect OOAD responsibility separation:
- `Controller` receives system operations.
- `OperatingMode` subclasses handle mode transitions.
- `ObstacleProcessor` and `DustProcessor` handle decision logic.
- Drivers own simulated hardware state.

- If the code differs from the original implementation prompt, follow the actual current `vibe/` headers and sources.

Required unit test count:
- `ControllerPowerButtonPressedTest`: 8 tests
- `ControllerStartButtonPressedTest`: 8 tests
- `ControllerDustDetectedTest`: 8 tests
- `ControllerTimerExpiredTest`: 6 tests
- `ControllerLowBatteryDetectedTest`: 7 tests
- `ControllerLowBatteryClearedTest`: 4 tests
- `ControllerChargeBatteryTest`: 9 tests
- `ControllerStopChargingTest`: 4 tests
- `ControllerObstacleDetectedTest`: 12 tests
- `ControllerClockTickTest`: 8 tests
- `OperatingModeTransitionTest`: 8 tests
- `ObstacleProcessorTest`: 8 tests
- `DustProcessorTest`: 4 tests
- `DriverStateTest`: 10 tests

Total: 104 Google Test unit tests.

Also add `vibe/tests/TEST_TRACEABILITY.md` mapping each suite to:
- related use case
- system/internal operation
- class/method
- expected mode or driver state.

If build integration is missing, add only files inside `vibe/`, such as `vibe/CMakeLists.txt`, so unit tests

answer result in Korean.

Part 3

: Unit Test

[Unit Tests]

104

[Test Suites]

14

[Vibe Test Pass]

100%

[검증 범위]

- Controller incoming operations: power/start/charge/dust/ obstacle/low battery/timer/clock tick	79
- OperatingMode transition: Standby, Normal, Boost, LowBattery	8
- Driver state: motor, cleaner, battery, sensor	12
- Processor decision: dust boost, obstacle avoidance direction	10

[Traceability]

root/vibe/tests/unit_tests/TEST_TRACEABILITY.md

Part 3

: System Test(Prompt)

You are Codex working on simulator-based black-box system tests for the RVC Control SW OOI assignment.

Scope:
- Treat `docs/` and `vibe/` as the current vibe-coding project artifacts.
- First inspect `docs/ai-output/srs.md`, `docs/ai-output/sdd.md`, and `vibe/simulator/main.cpp`.
- Do NOT edit files outside `vibe/`.
- Create scenario scripts under `vibe/system_tests/tc/`.
- Do NOT use Google Test for system tests.
- Use only the simulator CLI and `.rvc` scripts.

Task:
Implement broad but manageable simulator-based black-box system tests for the current `vibe/`
RVC Control SW implementation.

Required system test count:
- Positive scenarios: 12
- Negative scenarios: 30
- Total: 42
- Positive : Negative = 1 : 2.5

Required positive scenarios:
1. Power on enters `StandbyMode`
2. Power off from `StandbyMode` returns to `Off`
3. Start from `StandbyMode` enters `NormalMode`
4. Start from `NormalMode` returns to `StandbyMode`
5. Dust in `NormalMode` enters `BoostMode`
6. Timer in `BoostMode` returns to `NormalMode`
7. Low battery in `NormalMode` enters `LowBatteryMode` and stops motor
8. Low battery in `BoostMode` enters `LowBatteryMode` and stops motor
9. Charging is accepted in `StandbyMode`
10. Stop charging keeps system in a safe non-cleaning state
11. Obstacle front blocked and left clear keeps cleaning active
12. Obstacle all blocked keeps the system active while avoiding

Required negative scenarios:
1. Start while `Off` does not enter `NormalMode`
2. Dust while `Off` does not enter `BoostMode`
3. Dust while `StandbyMode` does not enter `BoostMode`
4. Dust while `LowBatteryMode` does not enter `BoostMode`
5. Timer while `Off` does not change mode
6. Timer while `StandbyMode` does not change mode
7. Timer while `NormalMode` does not change mode
8. Charge while `NormalMode` is rejected
9. Charge while `BoostMode` is rejected
10. Obstacle while `Off` does not start motor
11. Obstacle while `StandbyMode` does not start motor
12. Obstacle while `LowBatteryMode` does not start motor
13. Start while `LowBatteryMode` does not enter `NormalMode`
14. Repeated `start` in `StandbyMode/NormalMode` follows only valid toggle behavior
15. Repeated `power` operations only toggle on/off safely
16. Stop charging when not charging keeps charging off
17. Repeated stop charging remains safe
18. Low battery while `Off` is ignored
19. Repeated low battery in `LowBatteryMode` remains `LowBatteryMode`
20. Low battery cleared outside `LowBatteryMode` does not create invalid transition
21. Charge tick when not charging does not change mode
22. Clock tick while `Off` does not change mode
23. Clock tick in `StandbyMode` with no sensor event does not change mode
24. Clock tick in `NormalMode` with no sensor event does not change mode
25. Dust event while charging is ignored if current implementation blocks it
26. Obstacle with front clear in `StandbyMode` does not start motor
27. Obstacle all blocked in `StandbyMode` does not start motor
28. Timer after returning from `BoostMode` does not leave `NormalMode`
29. Charge after entering cleaning mode remains rejected
30. Unknown simulator command reports failure or unknown command without crashing, if supported

Each script must contain:
- input commands
- expectation commands
- observable PASS/FAIL output

Use existing simulator commands:
- `power`
- `start`
- `dust`
- `obstacle_front`
- `obstacle_left`
- `obstacle_right`
- `obstacle_all`
- `low_battery`
- `charge`
- `stop_charge`
- `timer`
- `charge_tick`
- `clock_tick`
- `low_battery_cleared`
- `expect_mode <name>`
- `expect_motor <on|off>`

If needed for stronger black-box coverage, add small expectation commands inside `vibe/simulator/main.cpp`:
- `expect_cleaner <on|off>`
- `expect_boost <on|off>`
- `expect_charging <on|off>`
- `expect_direction <Forward|Left|Right|Backward>`

Also add `vibe/system_tests/SYSTEM_TEST_TRACEABILITY.md` mapping each scenario to:
- SRS use case
- system operation
- positive/negative classification
- expected result.

answer result in Korean.

Part 3

: System Tests

[System Scenarios]	[Positive]	[Negative]	[Ctest Passed]
42	12	30	154/154

[Positive examples]

- P01 power on enters standby
- P03 start enters normal
- P05 dust enters boost
- P11/P12 obstacle avoidance

[Negative examples]

- start while off ignored
- charge while cleaning rejected
- dust while charging ignored
- unknown command no crash

Part 4

: Static Analysis

[Compiler warnings]

- vibe_rvc_core, simulator targets에 동일 적용
- GCC/Clang: -Wall -Wextra -Wpedantic
- MSVC: /W4

[Clang-tidy checks]

- performance-*, portability-*
- readability-*, modernize-*
- clang-diagnostic-*
- clang-analyzer-*
- bugprone-*

SA는 compiler warning과 clang-tidy rule set을 기준으로 정리했다.

Part 4

: Simulator

Script Simulator

- vibe/simulator/main.cpp
- script <file.rvc>
- system test 재현 인터페이스

expect_mode, expect_motor 등 검증 command

Map Simulator

- vibe/sim/app/main.cpp
- map <path>
- map/env/renderer 기반 interactive loop
- cleaned cell count, battery drain 표시

실행 예시

build-vibe\rvc_vibe_script_simulator.exe --script vibe/tests/system_tests/tc/P03_start_from_standby_enters_normal.rvc

build-vibe\rvc_vibe_map_simulator.exe --map vibe/sim/maps/default.map

Part 4

: Simulator - Preview

```
PS C:\Users\user\dev\cpp\OOAD\ooad\OOAD_TEAM1> .\vibe\build\rvc_simulator.exe
##########
#........#
#...#...##
#..#.....#
#......#.#
#......#.#
#...v..#.#
##########
tick 24 | mode BoostMode | battery 43% | motor moving BACK forward | cleaner on boost | cleaned 3/3
[p] power  [s] start  [c] charge  [q] quit
Cleaned 3/3 cells in 24 ticks
```

Part 3

: Simulator(Prompt) - simulator

You are a C++ simulator developer for the RVC (Robot Vacuum Cleaner) Control SW team project. The repository OOAD_TEAM1 is the source of truth. I am responsible for the simulator module.

Context
- The repo already contains:
- docs/ : SRS, SSD, sequence diagrams, domain model
- include/ , src/ : RVC core implementation (State pattern for operating modes, multi-file CMake project)
- tests/ : unit tests (Google Test)
- system_tests/tc/ : 43 system test cases
- Your work goes ONLY inside simulator/ .
- Do NOT modify files outside simulator/ unless explicitly asked.
- You MUST follow the existing code style, naming conventions, and State pattern usage already established in include/ and src/.

Task
Build a full-system simulator that models BOTH the RVC and its environment, so the RVC core code can be exercised end-to-end without real hardware.

The simulator must:
1. Model the environment:
- 2D grid room with configurable size
- Dust distribution (cleanable cells)
- Static obstacles (walls, furniture)
- Charging station location
- Battery drain / recharge dynamics
2. Drive the existing RVC core (from include/ and src/) as the System Under Test — instantiate real RVC classes, do NOT re-implement RVC logic inside the simulator.
3. Provide a time-step loop (e.g., tick()) that:
- Feeds sensor input to the RVC (dust detected, obstacle ahead, battery level, docking signal)
- Receives RVC commands (move, rotate, suction on/off, return-to-dock)
- Updates environment state accordingly
4. Expose a CLI entry point (simulator/main.cpp) that loads a scenario file (or hard-coded scenario for now) and prints per-tick state to stdout in a readable format.
5. Be runnable as its own CMake target (e.g., rvc_simulator), wired into the top-level CMakeLists.txt without breaking existing targets or tests.

Constraints
- C++17 (or whatever standard the existing CMakeLists uses — check first)
- No external dependencies beyond what's already in the project, except Google Test if used in simulator unit tests
- Prefer smart pointers over raw new/delete
- Keep environment and RVC decoupled — the simulator owns the environment and the RVC; the RVC must not know it's being simulated
- Build must pass on the existing toolchain

Process — follow this order
1. First, read and summarize:
- docs/ (especially SRS and SSD) to confirm RVC behaviors and sensor/actuator interfaces
- include/ and src/ to identify the public API of the RVC core (which methods to call as sensor inputs, which to query for actuator outputs)
- top-level CMakeLists.txt to understand how to add a new target
2. Propose a simulator file/class layout (Environment, Room, Sensor adapters, Actuator adapters, SimulationLoop, Scenario, main) and wait for my confirmation before writing code.
3. After I approve the layout, implement files one module at a time. Show diffs, not full file dumps, after the first version of each file.
4. Finally, add a minimal Google Test that runs a 50-tick scenario and asserts the RVC cleaned at least one cell.

Output rules
- Do not modify files yet in step 1 and step 2 — only summarize and propose.
- Do not invent RVC methods that don't exist in include/ . If something you need is missing, list it as a required-extension and ask me.
- Keep responses focused. No marketing language.

Approved. Proceed to implementation.

Reminders:
- Implement one module at a time. After each new file's first version, show me the diff (or full content if it's the first creation) and pause briefly so I can scan it.
- Order suggestion: CMakeLists.txt → Pose/Cell/DirectionUtil → Room → Environment → Scenario → RvcAdapter → SimulationLoop → main.cpp → SimulationLoopTest.cpp
- Stick to docking-free minimal implementation as agreed.
- If you discover the real Controller API needs something you didn't expect, stop and ask before working around it.

Part 3

: Simulator(Prompt) - Visualize(Real-time, Map)

Add terminal animation visualization to the simulator. Both
real-time mode and replay-from-recording mode must be supported.

Design constraints

- Visualization MUST be a separable concern. Introduce a Renderer
abstraction so the simulator core (Environment, SimulationLoop,
RvcAdapter) does NOT know whether it is rendering or not.
- Provide at least: NullRenderer (no output, for tests),
AnsiRenderer (terminal animation), and a recording mechanism.
- Tests under vibe/sim/tests/ MUST use NullRenderer. No sleeps,
no ANSI escapes, no terminal manipulation in tests.
- All new code stays under vibe/sim/ . Isolation rule still applies.

Real-time animation (AnsiRenderer)

- Each tick: clear the previous frame and redraw the room, robot,
dust, obstacles, charging station, and a HUD line.
- Use ANSI escape sequences only (no ncurses, no extra deps).
Hide cursor on start, restore on exit. Use cursor-home + redraw
rather than full screen clear if possible (less flicker).
- Glyphs (suggestion, adjust if you have better ideas):
. empty floor
* dust
# obstacle
C charging station
^ > v < robot (facing N/E/S/W)
- Color via ANSI:
dust = yellow, obstacle = gray, charging station = green,
robot = bright cyan, robot in BoostMode = bright magenta,
robot in LowBatteryMode = red.
- HUD shows: tick number, current mode name, battery level,
motor state, cleaner state, cells cleaned / total dust.
- Configurable tick delay via CLI flag --speed <ms> (default 100ms).

Windows compatibility

- On Windows, enable virtual terminal processing at program start
(SetConsoleMode with ENABLE_VIRTUAL_TERMINAL_PROCESSING on the
stdout handle) so ANSI escapes work in Windows Terminal /
PowerShell / cmd on Windows 10+. Wrap this in #ifdef _WIN32.
- On non-Windows, skip that and just emit ANSI.
- On program exit (and on Ctrl+C if feasible), restore cursor
visibility and reset colors.

Recording + Replay

- Add a Frame struct capturing everything needed to redraw one tick:
room dimensions, dust cells, obstacle cells, charging station,
robot pose, mode name, battery, motor/cleaner state, tick index.
- A FrameRecorder collects Frames during a run.
- After the run, optionally serialize frames to a file
(vibe/sim/include/sim/FrameLog.hpp). Use a simple line-based
text format — one frame per block, easy to diff and inspect.
Do NOT use JSON unless trivially hand-rolled; no new deps.
- A Replayer reads the file and feeds frames back into AnsiRenderer
at the configured speed. This path does NOT touch the real
Controller — it just renders prerecorded frames.

CLI

vibe/sim/app/main.cpp should accept:
--animate run live with AnsiRenderer (default on TTY)
--quiet run with NullRenderer (no output)
--speed <ms> per-tick delay for animation/replay
--record <path> write a FrameLog while running live
--replay <path> replay a FrameLog instead of running sim
--ticks <n> cap tick count

Implementation order

1. Frame struct + NullRenderer + Renderer interface
2. Wire SimulationLoop to call renderer.render(frame) each tick
3. AnsiRenderer (live drawing only, no recording yet)
4. FrameRecorder + FrameLog file format
5. Replayer + CLI wiring
6. Update existing GTest to use NullRenderer and assert correctness
7. Add one new GTest that records a 50-tick run to an in-memory
log and asserts frame count and final cleaned-cell count

Output rules

- Same as before: show diffs/new files one module at a time, pause
for me to review. Do not implement everything in one shot.
- If terminal capabilities or any cross-platform concern blocks you,
stop and ask. Don't pull in a dependency to "solve" it.

Part 3

: Simulator(Prompt) - simpler ver

Replace the previous visualization spec with this simpler one.
Drop everything not listed here.

What to keep
- Terminal animation only.
- Map loaded from a text file. Swap the file → simulator runs on a different map. Path passed via CLI: --map <path>
- All new code stays under vibe/sim/ . Isolation rule still applies.

What to remove from the earlier plan
- No recording, no replay, no FrameLog, no Replayer.
- No --quiet, --record, --replay flags.
- No charging station. Remove it from Room, Environment, Scenario, the renderer, and the map format. The simulator never models a charge point.
- Battery still drains per existing core rules; LowBatteryMode just ends activity. That stays. No recharge behavior.

Map file format

Plain text, one line per row, header line for size:

W H
<row 0>
<row 1>
...

Glyphs in the map file:
. empty floor
* dust
# obstacle / wall
R robot start position (exactly one required)

Rules:
- W and H are positive integers, space-separated, first line.
- Exactly H rows follow, each exactly W characters wide.
- Exactly one R. Robot starts facing North by default; an optional trailing char after R is not required for v1.
- Anything else is a parse error. On error: print a clear message with line number and exit non-zero. Do not crash.

Provide a default sample at vibe/sim/maps/default.map so it runs out of the box if --map is omitted.

Renderer

- Keep the Renderer interface so tests can use NullRenderer. (Justification: it's good OO design and required for clean tests; it is NOT user-facing complexity.)
- Only two implementations needed: NullRenderer (for tests), AnsiRenderer (the only one main.cpp uses).
- AnsiRenderer behavior:
- Each tick: cursor-home + redraw room + HUD line below.
- Hide cursor on start, restore on exit. Reset colors on exit.
- Colors:
dust = yellow, obstacle = gray, robot = bright cyan,
robot in BoostMode = bright magenta,
robot in LowBatteryMode = red.
- Robot glyph reflects facing direction: ^ > v
- HUD: tick, mode, battery, motor state, cleaner state, cleaned / total dust.

Windows compatibility

On Windows, enable virtual terminal processing at startup (SetConsoleMode + ENABLE_VIRTUAL_TERMINAL_PROCESSING on the stdout handle) inside #ifdef _WIN32. Otherwise emit ANSI directly.
Restore terminal state on normal exit.

CLI

vibe/sim/app/main.cpp accepts only:
--map <path> path to map file (default: vibe/sim/maps/default.map)
--speed <ms> per-tick delay (default: 100)
--ticks <n> max tick count (default: 500)

No other flags. If an unknown flag appears, print usage and exit.

Tests

- The existing 50-tick GTest uses NullRenderer and a small hard-coded or fixture-file map. It asserts at least one dust cell was cleaned.
- Add one map-parser test: feed a malformed map string and assert the parser reports an error rather than crashing.

Implementation order

1. Map file format + MapLoader + parser test
2. Renderer interface + NullRenderer
3. Wire SimulationLoop to call renderer each tick
4. AnsiRenderer (with Windows VT enablement)
5. main.cpp with the three CLI flags
6. Update existing 50-tick test to use NullRenderer

Same review rules as before: one module at a time, show diffs, pause for me to review. No new dependencies.

Part 3

: Simulator(Prompt) - buttons, battery

Update the simulator CLI and add interactive key controls. Apply these changes on top of the previous spec.

CLI simplification

Remove --speed and --ticks flags entirely. Hardcode the defaults:
tick delay = 200 ms (half of the previous 100 ms default)
tick cap = none (run until user quits or until all dust is cleaned, whichever comes first)

The only remaining flag is:
--map <path> default: vibe/sim/maps/default.map

Interactive key input during simulation

While the animation is running, the user can press keys to inject Controller events immediately (next tick at the latest). Treat this as polled non-blocking input checked once per tick, not as real OS signals. Key bindings:

p → controller.powerButtonPressed()
s → controller.startButtonPressed()
c → toggle charging:
- first press: controller.chargeBattery() and set an internal "charging active" flag
- while flag is set: call controller.chargingTick() every tick automatically
- second press: controller.stopCharging() and clear the flag
q → quit cleanly (restore terminal state, exit 0)

If the Controller rejects an event (e.g., charge in NormalMode), do NOT crash. Just ignore and continue — the HUD already reflects state, so the user will see nothing happened.

Non-blocking input implementation

Wrap platform-specific code behind a small KeyInput class under vibe/sim/include/sim/KeyInput.hpp and vibe/sim/src/KeyInput.cpp.

- Windows (#ifdef _WIN32): use _kbhit() + _getch() from <conio.h>.
- POSIX: put stdin in raw, non-canonical, non-echo mode via termios on construction, restore original termios on destruction (RAII). Use select() with zero timeout, or read() with O_NONBLOCK, to poll a single byte.
- Return std::optional<char> from KeyInput::poll(). Empty = no key this tick.

KeyInput must restore terminal state on every exit path, including exceptions and quit. AnsiRenderer's cursor restore + KeyInput's termios restore should both run via RAII so Ctrl+C doesn't leave the terminal broken — best effort.

HUD update

Add a second line under the existing HUD listing the hotkeys:
[p] power [s] start [c] charge [q] quit

Also reflect charging state in the HUD: when the internal "charging active" flag is set, append " CHARGING" to the mode field.

SimulationLoop changes

Per tick, in this order:
1. Poll KeyInput, dispatch any pressed key to the Controller (or set/clear the charging flag).
2. If charging flag is set, call controller.chargingTick().
3. Compute sensor inputs from environment, fire Controller events (dust, obstacle, low battery as before).
4. Read actuator state, update environment.
5. Build Frame, hand to Renderer.
6. Sleep 200 ms.

Exit conditions: KeyInput returned 'q', or all dust cells cleaned. Print a final summary line after the loop ends ("Cleaned N/M cells in T ticks").

Tests

KeyInput must be injectable. Make SimulationLoop take a KeyInput interface (or std::function<std::optional<char>()>), so tests can feed a scripted key sequence and run with NullRenderer + zero sleep.

Add one new test: script the sequence [p, s] at ticks 0 and 1, run 50 ticks on a small map, assert at least one dust cell cleaned and the final mode is not Standby.

The existing 50-tick test continues to use an empty key source (always returns nullopt).

Same review rules: implement one module at a time, show diffs, pause for review. No new third-party dependencies.

: 장단점 및 소감

장점

개발 속도 / 생산성
산출물 간 일관성
학습 효과 / 진입 장벽
테스트 / 품질
문서화

단점

환각(Hallucination) / 정확성
설계 깊이 부족
디버깅이 더 어려워짐
의존성 / 환경 문제
평가 / 책임 문제

소감

가장 인상적이었던 점은 속도, 일관성이었다. AI에 이전 단계 산출물을 컨텍스트로 누적시키며 작업하니, 용어와 구조가 자연스럽게 통일되었고 개발 산출물이 하나의 일관된 흐름으로 이어진다는 느낌을 받았다.

감사합니다