LLM을 활용한 SW 리팩토링

팀원: 김병현, 박준하

지도교수: 채흥석

부산대학교 정보컴퓨터공학부

연구 배경

LLM 발전

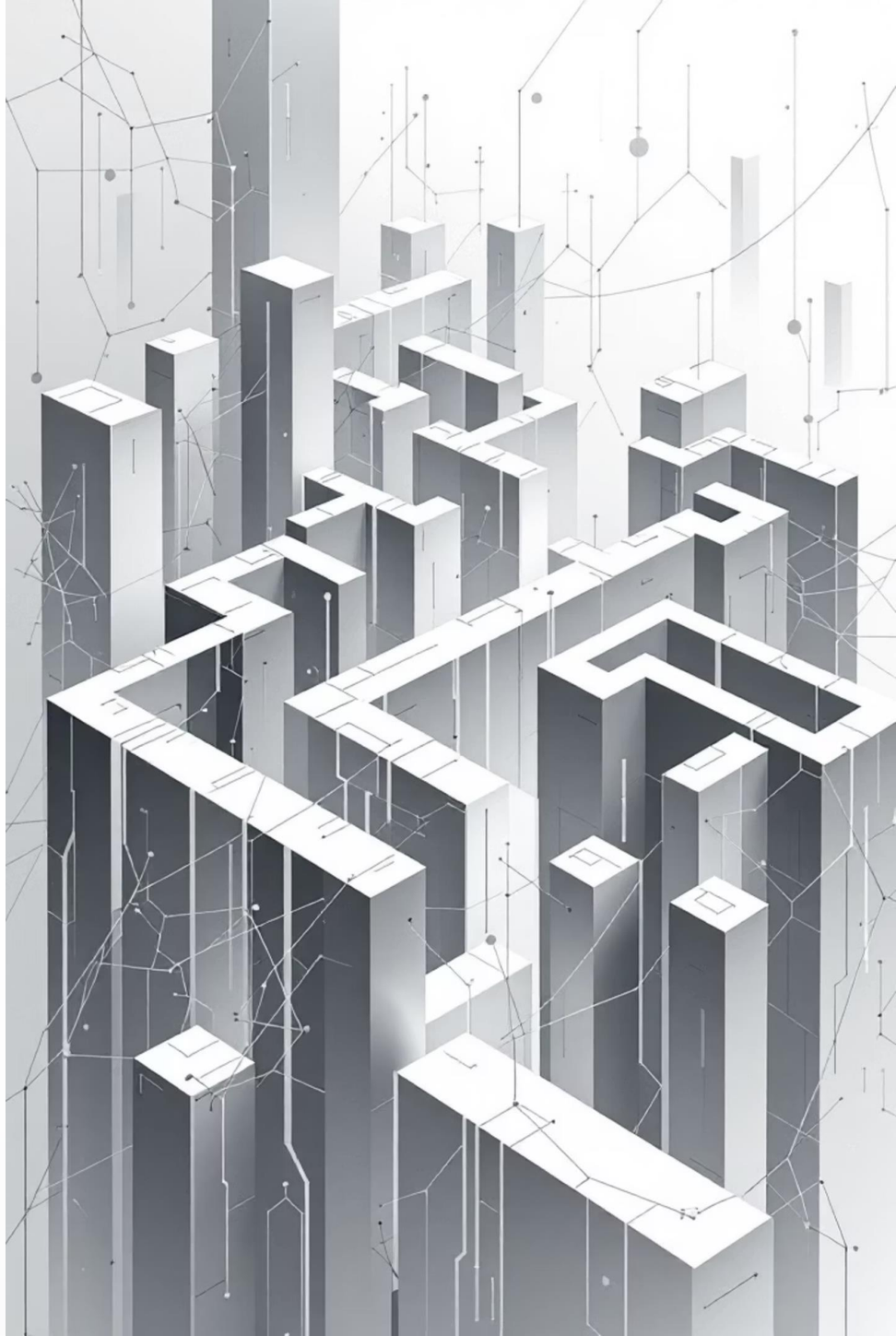
GPT, Gemini 등 LLM이 코드 생성, 디버깅을 넘어 리팩토링까지 활용 가능

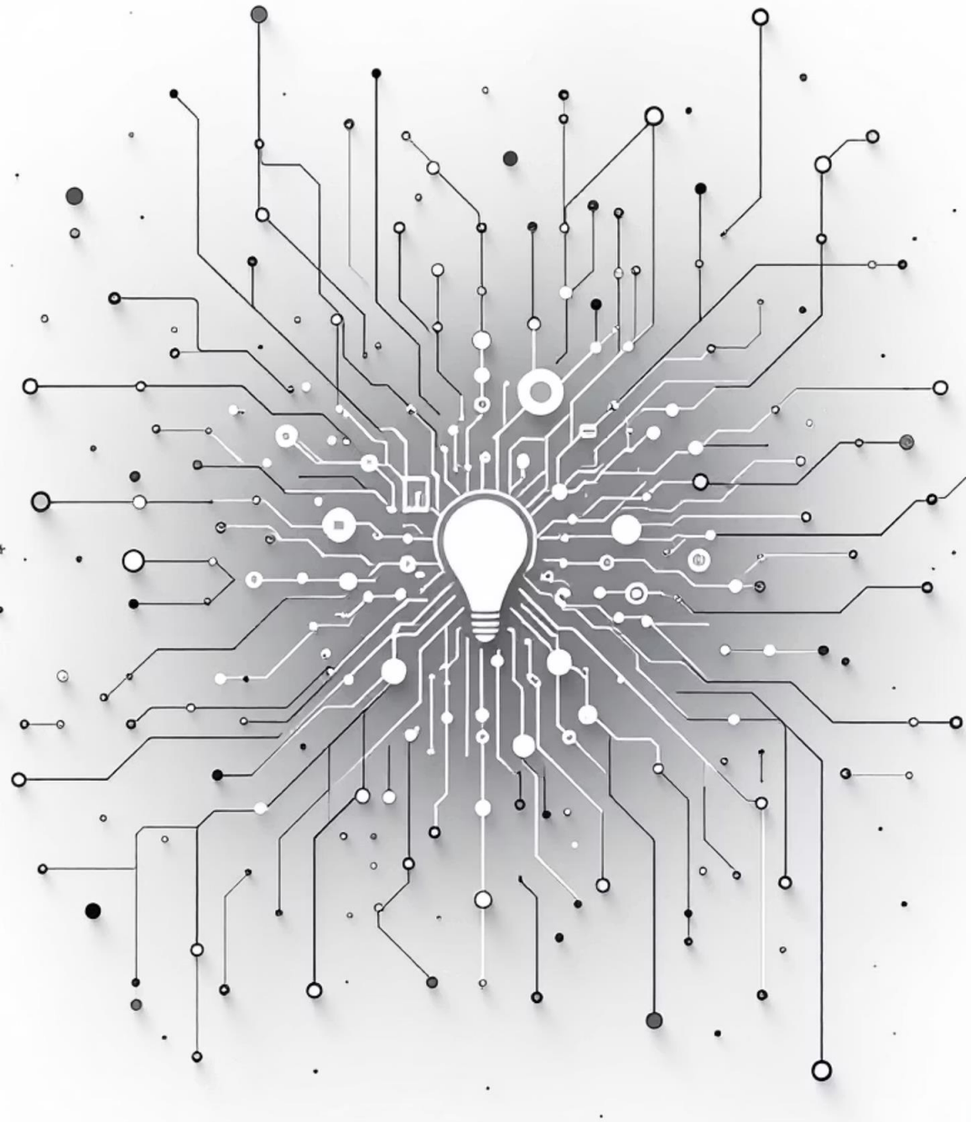
유지보수성 중요성

SW 생명주기 비용의 **50~80%**가 유지보수에 소요

기술 부채 문제

낮은 유지보수성이 개발 속도 저하 및 버그 발생 증가 유발





연구 목표

LLM을 활용해 SW 공학의 고질적 문제인 '유지보수성 저하'를 해결하고자 함

연구 질문

01

유지보수성 개선 능력

LLM을 활용한 리팩토링이 코드의 유지보수성을 의미 있게 개선하는가?

02

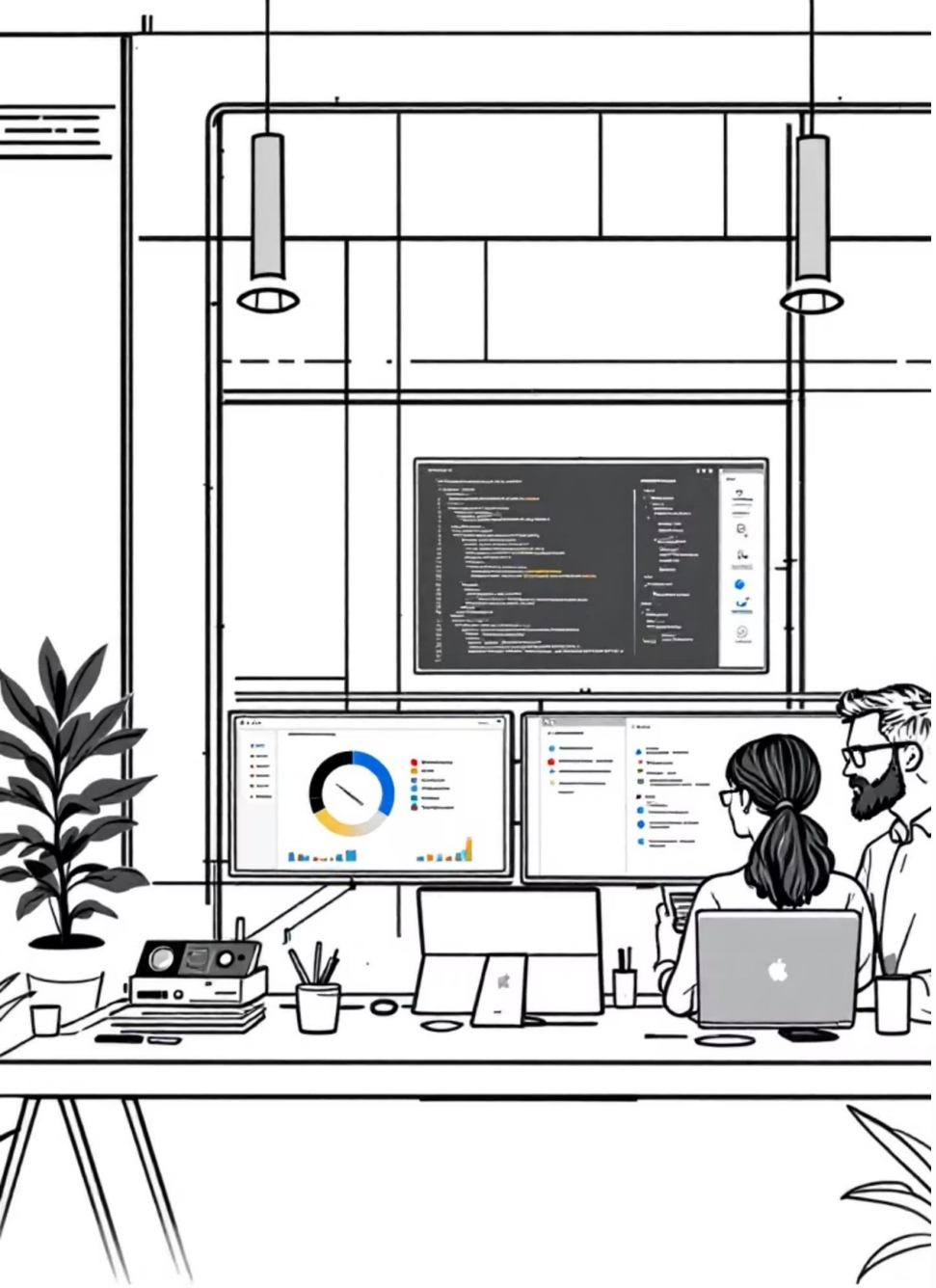
코드 특성과의 관계

코드 규모(LOC), 복잡도(CC)가 LLM의 리팩토링 효과에 어떤 영향을 미치는가?

03

연속적인 개선 능력

동일한 코드에 리팩토링을 반복 적용하면 유지보수성이 안정적으로 향상되는가?



실험 설계

실험 대상 LLM

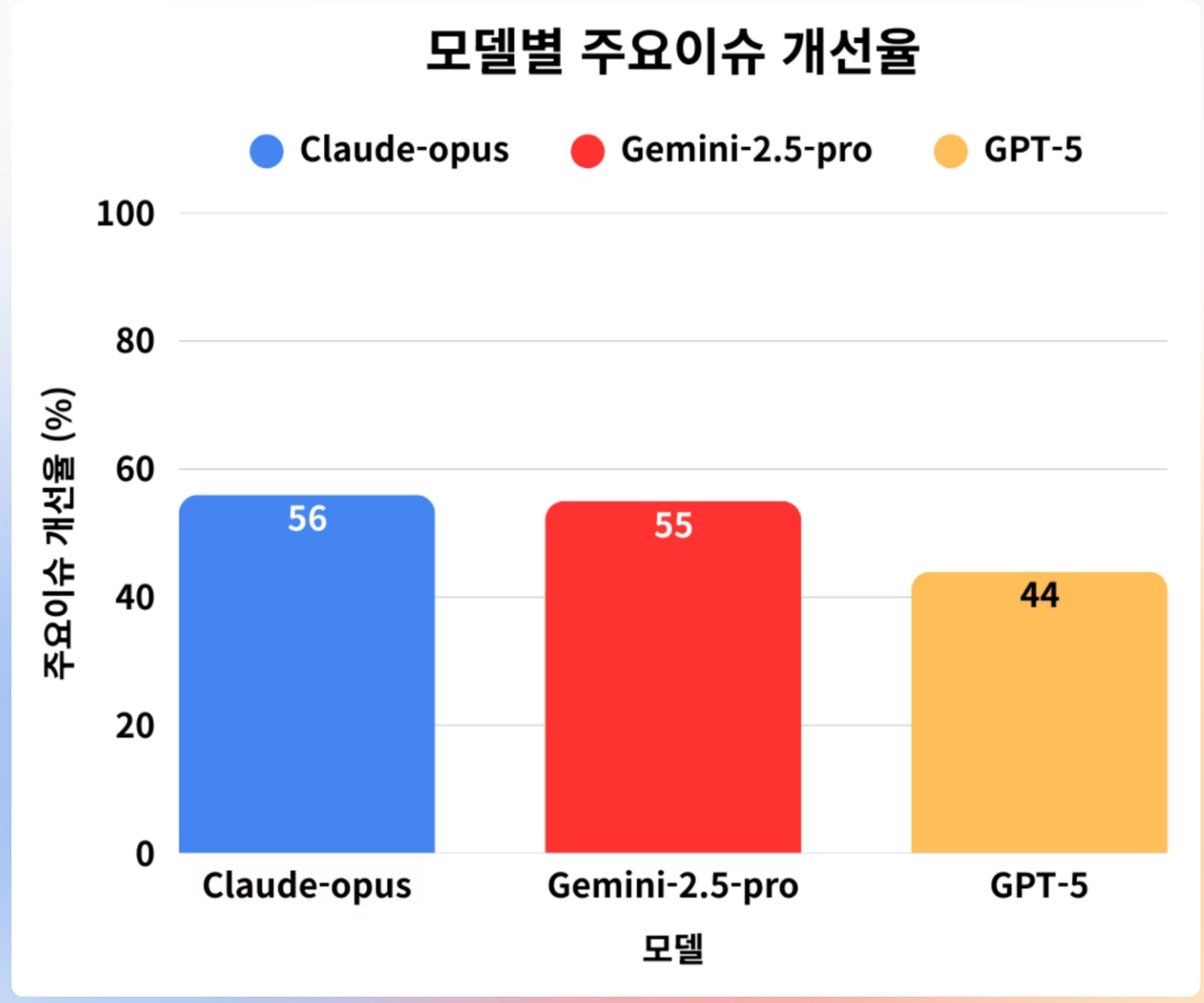
- Gemini-2.5-pro
- GPT-5
- Claude-Opus-4-1

데이터셋

C언어 오픈소스 SW

- MINIX
- Coreutils
- glibc
- binutils-gdb
- Lua

평가 방법: SonarQube Cloud를 사용한 정적 분석으로 유지보수성 이슈 수 측정



유지보수성 개선 능력 결과

55.94%

Claude-opus-4-1

가장 높은 개선율

55.12%

Gemini-2.5-pro

안정적인 성능

43.79%

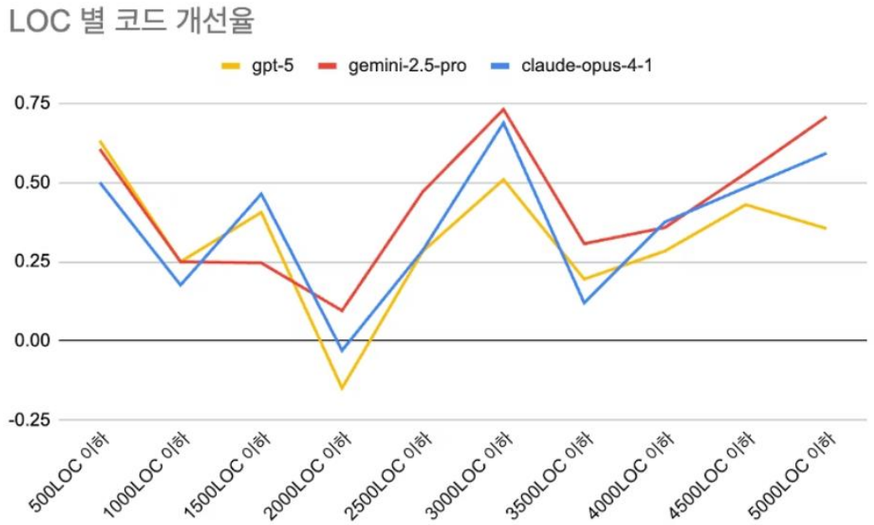
GPT-5

기본적인 개선 효과

3가지 LLM 모두 심각도 높은 주요 이슈(Blocker, High) 개선에 효과적

코드 특성과의 관계

코드 규모(LOC) 1

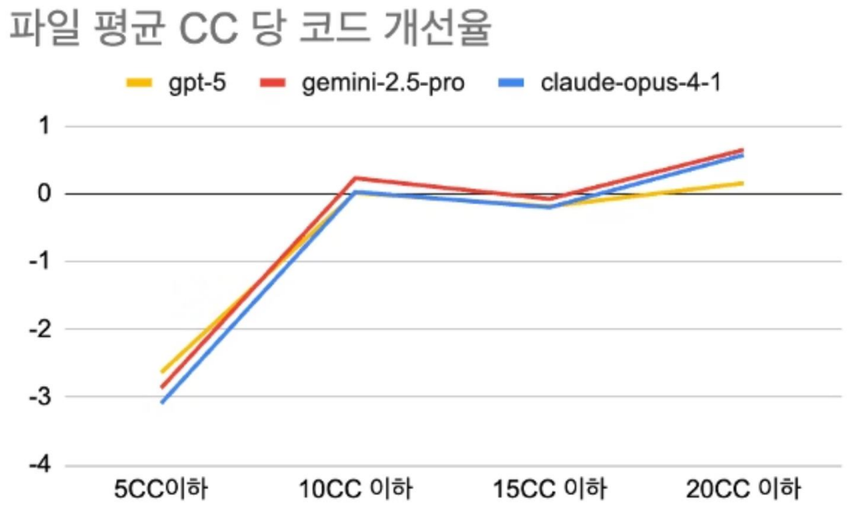


~2000 LOC 근방에서 성능 저하 발생 ('인지적 과부하' 추정)

전체적으로 균일하지는 못하다

2 코드 복잡도(CC)

낮은 복잡도(5CC 이하): 불필요한 수정으로 품질 악화 가능



높은 복잡도(10CC 이상): LLM 개선 효과 극대화

LLM 리팩토링은 복잡한 모듈에 선별적으로 적용할 때 가장 효율적

연속 개선 능력

1 초기 단계 (1~2회차)

개선 효과가 집중되는 시기

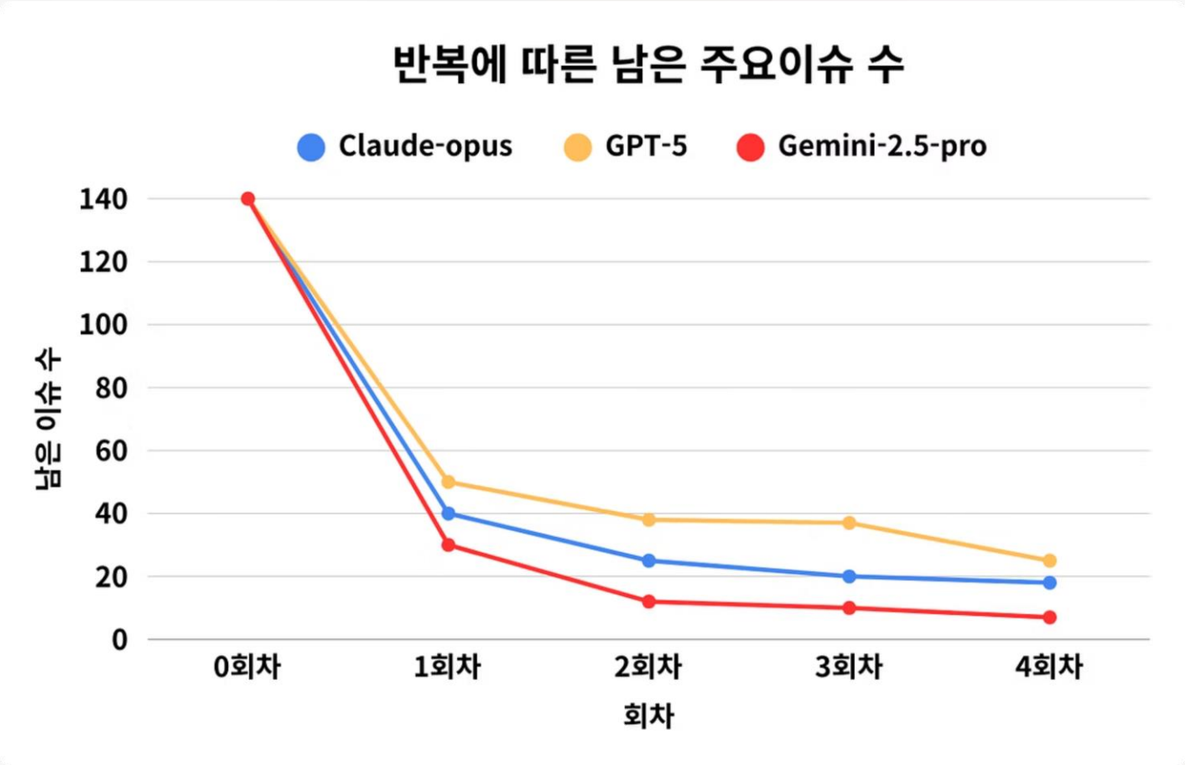
2 중간 단계

점진적 수렴 패턴 시작

3 후기 단계

해결하기 어려운 이슈만 잔존

기술적 한계: 깊은 중첩 구조와 같이 복잡도가 매우 높은 코드는 LLM이 수정을 거부하는 경향



결론

유의미한 개선

LLM은 C언어 코드의 유지보수성을 유의미하게 개선 가능

선별적 전략

복잡한 코드에 집중하는 선별적 적용이 중요

트레이드오프

주요 이슈 해결 시 부차적 이슈 발생 가능

> LLM은 C언어 레거시 코드, 특히 복잡하고 위험도 높은 코드를 개선하는 강력한 도구

향후 연구 방향

기능적 정확성 검증

자동화된 단위/회귀 테스트 파이프라인 통합

에이전트 기반 전략

분석-전략-수정-검증을 수행하는 AI 에이전트 연구

인간-AI 협업

LLM 제안과 인간 개발자 선택의 협업 모델

아키텍처 리팩토링

모듈 간 의존성 분석 등 거시적 리팩토링 확장