W07
컨텍스트 구조화

맥락이 정확해야 개선이 된다

추가로 더 개발을 하면 발생하는 일

"다음에 알림 기능을 추가할 거야. 이 기능을 추가하기 위한 PRD를 만들어줘"

- AI는 최대한 서비스 맥락을 이해하려고 노력하지만, 우리만큼 이 서비스에 관심이 없다.

실제로 일어나는 일:

- 기존에 있는 컴포넌트를 활용하지 않고 새로 만든다
- 이미 정의된 API 패턴을 무시하고 다른 방식으로 호출한다
- 기존 폴더 구조를 무시하고 엉뚱한 위치에 파일을 생성한다
- 새 기능은 동작하지만 기존 기능이 깨진다
- 용어가 뒤섞인다 (user/member/account가 혼용)

"AI는 코드를 읽을 수 있지만, 의도를 읽지는 못한다"

‘바이브 코딩'의 화두 – ‘맥락'을 어떻게 유지할 것인가?

2025년 2월, Andrej Karpathy가 '바이브 코딩'이라는 용어를 만들었다

- "코드가 존재한다는 사실조차 잊고, 완전히 바이브에 몸을 맡겨라"
- 바이브 코딩은 자연어로 원하는 것을 설명하면 AI가 코드를 생성해주는 방식이다. 프로그래머의 역할이 직접 코딩에서 AI가 생성한 코드를 안내하고, 테스트하고, 피드백을 주는 것으로 바뀐다.

그리고 2025년 하반기, 업계는 '바이브 코딩의 숙취'를 경험했다

- 9월, Fast Company는 시니어 개발자들이 AI가 생성한 바이브 코드로 작업할 때 '개발 지옥'을 겪고 있다고 보도했다.

'바이브 코딩'의 화두 - '맥락'을 어떻게 유지할 것인가?

문제의 핵심: 맥락(Context)의 부재

- 사용자들이 더 많은 것을 요구하고 프롬프트가 점점 커졌지만, 모델의 신뢰성은 오히려 흔들리기 시작했다.

바이브 코딩에서 '컨텍스트 엔지니어링'으로

- 2025년은 AI 소프트웨어 엔지니어링 활용에서 중대한 전환이 일어났다. 느슨한 바이브 기반 접근에서 AI가 맥락을 처리하는 방식을 체계적으로 관리하는 접근으로 바뀌었다.
- Shopify CEO Tobi Lutke: "나는 프롬프트 엔지니어링보다 컨텍스트 엔지니어링이라는 용어가 정말 좋다." Karpathy도 동의: "프롬프트 엔지니어링보다 컨텍스트 엔지니어링에"

전체 코드를 검토시키는 것은 무의미 하다

"프로젝트 전체 코드를 분석해서 파악해줘"

왜 안 되는가:

- 컨텍스트 윈도우 한계: 파일이 많으면 전부 읽지 못한다
- 우선순위 부재: 중요한 것과 덜 중요한 것을 구분하지 못한다
- 의도 파악 불가: 코드가 "왜" 이렇게 작성되었는지 모른다
- 휘발성: 다음 대화에서 다 잊어버린다

자동 컨텍스트는 한계가 있다

커서의 RAG를 활용한 맥락 관리

- 장점: 코드베이스에서 관련 파일을 자동으로 찾아준다
- 한계점: "관련성"을 키워드로 판단하므로 구조적 맥락을 놓친다

문서 참조(@docs)를 통한 맥락 관리

- 장점: 명시적으로 문서를 지정하면 확실히 읽는다
- 한계점: 문서가 없으면 참조할 것이 없다, 문서가 오래되면 현실과 맞지 않다

그 외의 코딩 에이전트들이 제공하는 도구들

- Codebase indexing, Symbol search 등
- 개발자들에게 "코드를 찾는" 도구 - "서비스의 맥락"을 유지시켜 주지는 못한다

주기적으로 맥락을 관리하고 재조정해야 한다

도구가 해결해주길 기다리지 말고, 맥락 문서를 직접 만들어야 한다

언제 맥락을 정리해야 하는가?

- 마일스톤을 완료했을 때
- 새로운 기능을 추가하기 전
- 리팩토링을 완료한 후
- 오랜만에 프로젝트를 다시 열었을 때

왜 "주기적"이어야 하는가?

- 한 번 정리하고 끝이 아니다
- 코드는 계속 변한다 → 문서도 따라가야 한다
- 오래된 맥락 문서는 오히려 AI를 혼란시킨다

관리가 필요한 맥락

1. 기술 맥락: 이 프로젝트는 "무엇으로" 만들어졌는가
   - "새 API는 어디에 만들어야 해?" → 폴더 구조, 아키텍처 참고
2. 도메인 맥락: 이 서비스는 "무슨 개념"을 다루는가
   - "주문 취소가 가능한 조건이 뭐야?" → 비즈니스 규칙 참고
3. 서비스 맥락: 이 서비스는 "어떻게 동작"하는가
   - "결제 완료 후 어느 화면으로 가야 해?" → 화면 연결 참고
4. 디자인 맥락: 이 서비스는 "어떻게 보여야" 하는가 (다음 시간에)

1-1. 기술 스택

프로젝트에서 사용하는 언어, 프레임워크, 라이브러리 목록

- AI가 새 코드를 작성할 때 이미 설치된 라이브러리를 활용하도록
- 프로젝트와 맞지 않는 기술을 제안하지 않도록
- 버전 호환성 문제를 방지하도록

"설정 파일들과 코드 전체를 분석해서 사용 중인
언어/프레임워크/라이브러리/인프라 등의 기술 스택을 기술명, 버전, 이
프로젝트에서의 용도 등 어떤 이유와 목적으로 어떻게 사용되고 있는지를 목록으로
정리해줘"

1-2. 아키텍처

코드의 구조적 설계 - 레이어 구분, 모듈 간 의존 방향, 데이터 흐름

- 새 기능 추가 시 어느 레이어에 코드를 작성해야 하는지 알려주기 위해
- 기존 패턴을 따르도록 가이드하기 위해
- 의존성 방향을 유지하기 위해

“현재 프로젝트의 폴더 구조와 주요 파일들을 분석해서 아키텍처 패턴을 파악해줘. 사용 중인 아키텍처 패턴 이름과 레이어/모듈의 구분, 데이터 흐름, 모듈간 의존성 규칙 등 이 코드의 일관성과 확장성, 유지보수성을 유지하기 위한 아키텍처들을 실제 코드에서 발견해 이를 기반으로 작성해줘”

1-3. 폴더 구조

파일과 디렉토리 배치 규칙 - 각 폴더의 역할과 파일 배치 기준

- AI가 새 파일을 만들 때 올바른 위치에 생성하도록
- 기존 파일을 찾을 때 어디를 봐야 하는지 알려주기 위해
- 프로젝트 일관성을 유지하기 위해

"프로젝트 루트의 폴더 구조를 분석해서 각 폴더의 역할을 정리해줘. 새 파일을 만들 때 어디에 위치해야 하는지 판단할 수 있게 정리해줘. 현재 구조에서 일관성이 깨진 부분이 있다면 별도로 표시해줘."

1-4. 외부 제약

기술적으로 불가능하거나 금지된 것 - API 한계, 인프라 제약, 건드리면 안 되는 코드

- AI가 불가능한 방법을 제안하지 않도록
- 건드리면 안 되는 부분을 보호하기 위해
- 시간 낭비를 줄이기 위해

“README, 주석, 설정 파일 및 프로젝트 전체를 검토해서 사용 중인 외부 서비스의 제약사항, 인프라 한계, 수정하면 안 되는 코드 영역을 정리해줘.”

DDD(Domain Driven Design)이란?

도메인 = 이 서비스가 다루는 "비즈니스 영역"

- 쇼핑몰이라면: 상품, 주문, 결제, 배송, 회원
- 병원 예약이라면: 환자, 의사, 예약, 진료, 처방

DDD의 핵심 아이디어:

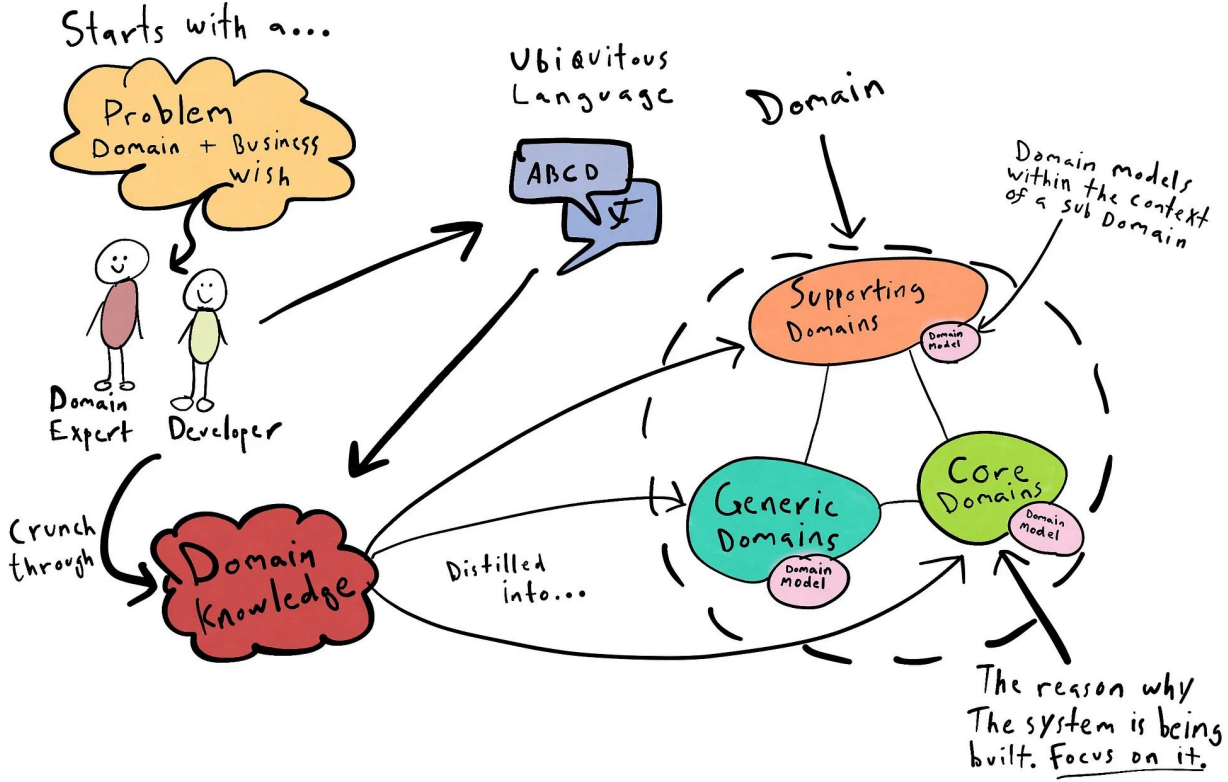
- "코드를 기술 중심이 아니라 비즈니스 중심으로 설계하자"
- 기술 중심: UserController, UserService, UserRepository
- 비즈니스 중심: 회원가입, 로그인, 프로필관리

비개발자에게 DDD가 필요한 이유:

- AI에게 "주문 취소 기능 만들어줘"라고 했을 때
- 도메인 맥락이 없으면: 그냥 DB에서 삭제하는 코드 작성
- 도메인 맥락이 있으면: "결제 완료 후 24시간 이내만 취소 가능" 규칙 반영

"AI가 비즈니스 규칙을 알아야 올바른 코드를 작성할 수 있다"

DDD(Domain Driven Design)이란?



2-1. 용어 정의

프로젝트에서 쓰는 비즈니스 용어와 그 의미

- AI와 사람이 같은 단어를 다르게 이해하는 것을 방지
- 코드에서 일관된 네이밍을 유지하기 위해
- 모호한 요청을 명확하게 만들기 위해

“코드에서 사용되는 도메인 용어들을 추출해줘. 타입 정의, 변수명, 주석에서 비즈니스 개념을 나타내는 단어들을 찾아서 각각의 의미를 정리해줘.”

2-2. 엔티티와 관계

핵심 데이터 객체와 연결 - 엔티티 목록, 속성, 엔티티 간 관계

- 새 기능이 기존 데이터 구조와 어떻게 연결되는지 파악하기 위해
- DB 스키마 변경 시 영향 범위를 알기 위해
- AI가 올바른 관계를 가진 코드를 작성하도록

"프로젝트의 타입 정의, 스키마, 모델 파일을 분석해서 핵심 엔티티를 추출해줘. 각 엔티티의 목록과 각각의 주요 속성, 엔티티 간 관계(1:1, 1:N, N:M)를 정리해줘."

2-3. 비즈니스 규칙

도메인에서 허용/금지되는 것 - 조건, 제약, 계산 로직, 상태 전이 조건

- AI가 "기능은 되지만 규칙에 맞지 않는" 코드를 작성하지 않도록
- 엣지 케이스를 놓치지 않기 위해
- 비즈니스 로직의 일관성을 유지하기 위해

“코드에서 조건문, 유효성 검사, 에러 처리 로직을 분석해서 비즈니스 규칙을 추출해줘. "~하면 안 된다", "~일 때만 가능하다", "~는 ~여야 한다" 형태로 정리해줘. 규칙과 함께 해당 내용의 코드 위치도 명시해줘”

3-1. 화면 목록과 구조

서비스를 구성하는 화면들 - 화면 이름, 역할, 계층

- 새 기능이 어느 화면에 들어가야 하는지 파악하기 위해
- 화면 간 중복을 방지하기 위해
- 전체 서비스 구조를 AI에게 알려주기 위해

“프로젝트 폴더의 구조를 분석해서 화면 목록을 추출해줘. 각 화면의 경로, 이름, 역할을 계층 구조로 정리해줘.”

3-2. 기능

서비스가 제공하는 단위 행위 - 기능명, 설명, 연결된 화면, 관련 도메인 규칙

- 새 기능이 기존 기능과 어떻게 연결되는지 파악하기 위해
- 중복 개발을 방지하기 위해
- 기능 변경 시 영향 범위를 알기 위해

“각 화면에서 사용자가 수행할 수 있는 주요 행위를 분석해줘. 화면을 중심으로 실제 UI 동작을 검토해 기능명, 연결된 화면, 설명, 관련된 비즈니스 규칙을 정리해줘.”

3-3. 화면 간 연결

화면 이동 규칙 - 라우팅 경로, 네비게이션 흐름

- 새 기능 추가 후 어느 화면으로 이동해야 하는지 알기 위해
- 사용자 흐름의 일관성을 유지하기 위해
- 누락된 연결을 발견하기 위해

“코드에서 화면 이동 로직을 분석해서 화면 간 연결을 정리해줘. 주요 사용자 흐름과 조건부 이동 규칙 등 어떤 조건이나 액션에서 어디로 이동하는지 흐름을 정리해줘.”

맥락을 체계적으로 관리하려면?

문서를 이용한다 (.md, .yaml)

- 장점: 사람이 읽고 수정하기 쉽다, 별도 도구 없이 텍스트 에디터로 편집
- 단점: 문서가 오래되면 현실과 맞지 않을 수 있다, 여러 문서 간 일관성 유지가 어렵다

DB를 이용한다

- 장점: 구조화된 쿼리로 조회 가능, 일관성 유지가 쉽다
- 단점: 설정이 복잡하고 오버 엔지니어링이 될 수 있다, AI에게 직접 참조시키기 어렵다

문서로 시작하고, 필요하면 DB를 활용해 자동화

오늘 해결해야 할 문제

1. 프롬프트를 사용해 현재 프로젝트의 기술 스택, 아키텍처, 폴더 구조, 외부 제약을 정리한다
1. 프롬프트를 사용해 용어 정의, 엔티티와 관계, 비즈니스 규칙을 정리한다
2. 프롬프트를 사용해 화면 목록, 기능, 화면 연결, 권한을 정리한다
3. agents.md에 맥락 문서 위치와 참조 방법을 추가한다
4. (선택) 맥락 문서를 참조시킨 상태에서 다음 로드맵 기능을 요청해본다

Q&A