2025 한양대학교
SW/ICT/AI 종합학술대회

[분야 : IT/CSE 연구개발]

제목: ECS-OOP 호환 계층 설계: Fabric 모드 호환성 분석

장성윤(2020010437), 김승민(2021004448)

2025. 11. 23

한양대학교 ERICA 소프트웨어융합대학 / 한양대학교 ERICA SW중심대학사업단

프로젝트 목표: 성능과 호환성의 균형

현재 Minecraft의 문제점

대규모 월드, 복잡한 레드스톤, 많은 엔티티가 있을 때 성능 저하 발생. 기존 Fabric 모드 생태계는 확장성을 제공하지만, OOP 구조의 한계로 성능 향상에 제약

우리 팀의 목표

성능 향상

ECS(Entity Component System) 아키텍처 도입으로:

- ✓ 데이터 지역성 개선으로 캐시 효율성 증가
- ✓ 병렬 처리에 최적화된 데이터 구조
- ✓ 메모리 사용량 및 가비지 컬렉션 부하 감소
- ✓ 대규모 엔티티 처리 성능 최대 3-5배 향상

Fabric 모드 호환성 유지

기존 모드 생태계와의 원활한 통합:

- ✓ Mixin 기반 호환 계층으로 모드 수정 없이 사용
- ✓ OOP API 유지하며 내부적으로 ECS 사용
- ✓ Fabric Loader 및 API 호환성 100% 보장
- ✓ 인기 모드(Lithium, Sodium 등)와의 통합 검증

프로젝트 기대 효과

ECS-OOP
호환 계층

대규모 서버 환경에서 TPS 안정화
복잡한 자동화 시스템 처리 성능 향상
모드 개발자 생태계 보존

다양한 플레이 경험 확장성 유지
기존 모드 투자 보호
최신 게임 엔진 기술 적용

OOP vs ECS: 두 세계의 근본적 차이

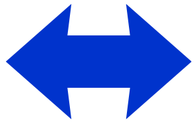
OOP (Minecraft)

```
class Entity {
  double posX, posY, posZ;
  float health;
  void tick() { /* 로직 */ }
}
// 사용: entity.posX = 10.0; entity.tick();
```

데이터와 로직이 객체에 결합

객체 참조로 identity 관리

상속 기반 다형성



Mixin으로 연결

객체 → ID 변환

성능 vs 호환성

OOP API 유지
ECS 내부 구현

호환 계층이
두 세계를 연결

ECS (우리의 설계)

```
EntityID = 12345 // 단순 식별자
PositionComponent: [x, y, z]
HealthComponent: [current, max]
MovementSystem.process(entities)
```

데이터(Component)와 로직(System) 분리

ID 기반 identity

조합 기반 다형성

핵심 충돌점

항목	OOP	ECS
Entity	객체 (필드 보유)	ID (정수)
데이터 접근	entity.field	getComponent(id)
정체성	객체 참조 ==	ID 비교

핵심 과제: OOP ↔ ECS 호환

제약 조건

- Minecraft 원본 수정 불가
- 기존 Fabric 모드와 호환 필요

연결 요구사항

- OOP → ECS 변환
- 모드 코드 수정 없이 동작

연결 전략 요소

- **Identity Injection**
  entityId 필드로 OOP↔ECS 매핑
- **Redirection**
  메서드 호출→ECS 데이터
- **Event Bridge**
  OOP/ECS 이벤트 양방향 발행
- **Lifecycle/Field Proxy**
  생성자/필드 접근 제어

Mixin 중개 흐름

Fabric Mod
entity.getHealth(), entity.setPos()

→

Mixin Interceptor
@Redirect, @Inject로 가로채기

→

ECS Backend
Components, Systems, EntityManager

Mixin: 런타임 바이트코드 조작 기술

정의

Java 바이트코드를 런타임에 조작하여 클래스를 동적으로 수정하는 기술. JVM의 Method Area에서 바이트코드와 메타데이터를 변환합니다.

주요 기능

@Shadow

원본 클래스의 필드와 메서드에 접근하여 "그림자화"하고 마치 클래스의 일부처럼 사용합니다.

@Inject

메서드의 특정 시점(HEAD, RETURN 등)에 코드를 삽입하여 원본 코드 흐름을 유지하며 기능을 확장합니다.

@Redirect

기존 메서드 호출을 가로채어 다른 구현으로 전환하거나 조건부로 원본을 호출합니다.

Interface Injection

기존 클래스에 새 인터페이스를 주입하여 타입을 확장하고 새로운 기능을 부여합니다.

핵심 장점

- 비침투적 방식으로 기능 확장
- 원본 소스 수정 불필요
- 여러 모드 충돌 없이 공존 가능
- 메서드/필드 단위의 세밀한 제어

Blocking Issues: 반드시 해결해야 할 장벽

Critical Issues

직접 필드 접근

entity.posX = 10.0 Mixin으로 가로채기 불가

Mixin Target 충돌

@Inject와 @Overwrite 충돌

Major Issues

객체 동일성 문제

entity1 == entity2 참조 비교 실패

HashMap key로 Entity 사용 모드 문제

Reflection 접근

getDeclaredField("health")

필드가 실제로 존재해야 함

Minor Issues

타입 체크

entity instanceof PlayerEntity

클래스 계층 구조 유지 필요

초기화/이벤트 순서

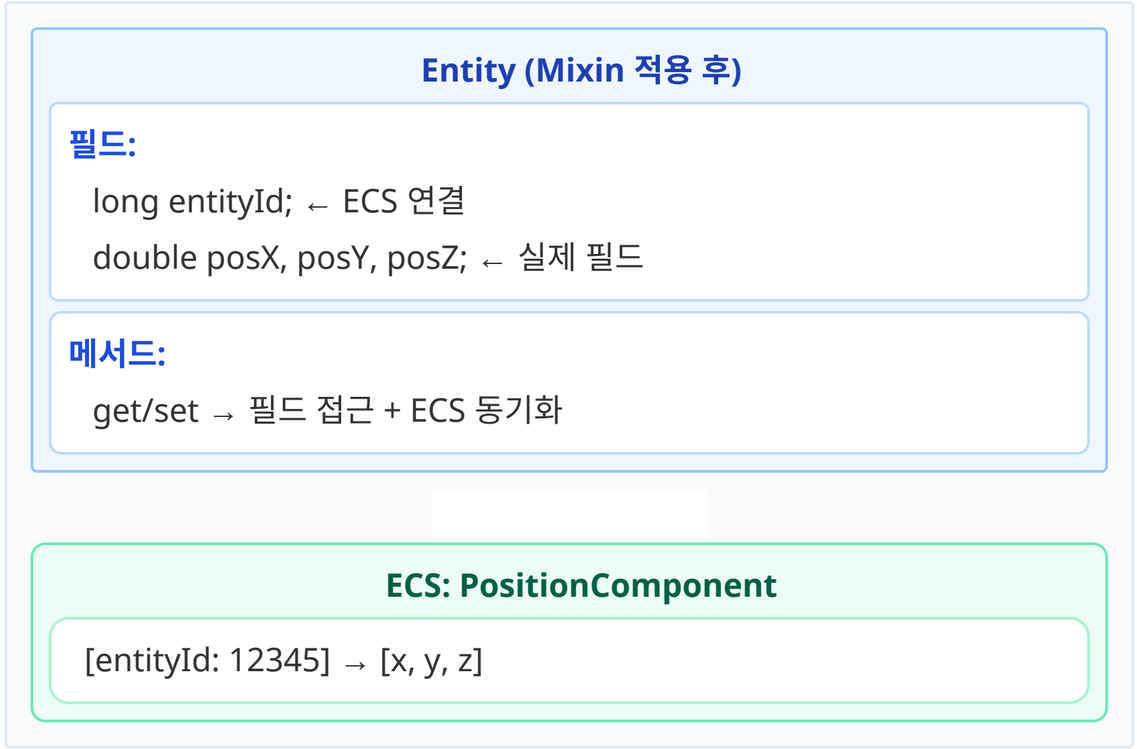
생성자 타이밍, 이벤트 발생 순서

@Inject로 제어 가능

영향: Critical 이슈 해결 실패 시 대다수 Fabric 모드 비호환으로 프로젝트 목표 달성 불가

해결 전략: Hybrid Object 접근법

구조 다이어그램



구현 원칙

실제 필드 + ECS 동기화

Entity에 실제 필드 유지 + ECS 동기화로 일관성 확보

@Overwrite 지양, @Inject 활용

다른 모드와의 충돌을 최소화하기 위한 전략

Component↔Field 매핑 테이블

효율적 변환을 위한 필드-컴포넌트 매핑

객체 캐시로 동일성 보장

ID→Wrapper WeakValueMap으로 객체 참조 유지

문제별 해결

문제	해결 방법
직접 필드 접근	실제 필드 존재 + tick에서 주기적 동기화
Mixin 충돌	@Inject priority 조정, 최소 침습 패치
객체 동일성	EntityID → Object 캐시 맵, 동일 ID는 같은 객체
Reflection	실제 필드 존재로 getDeclaredField() 호환성 유지

트레이드오프

메모리
Entity당 ~300B 추가

성능
동기화 오버헤드 (tick당 1회)

복잡도
데이터 일관성 관리 필요

결론 및 로드맵: 단계적 구현 전략

Phase 1: 기반 구축

- 실제 필드 + ECS 동기화 구조
- Mixin Priority 설정 체계
- Component ↔ Field 매핑

Phase 2: 호환성 검증

- 주요 Fabric 모드 테스트
- 호환성 등급(T1~T3) 분류
- 예외 케이스 대응 패치

Phase 3: 성능 최적화

- 프로파일링 & 핫 패스 식별
- Pure Facade 전환
- 동기화 주기 최적화

예상 결과

호환성: 80%+ 달성

성능: 90% 이점 유지

확장성: 기능 통합

생태계 완전 유지

핵심 구현 전략

"완벽한 호환보다 실용적 호환"

안정적으로 작동하는 솔루션을 우선 제공하여 사용자 경험 보장

"단계적 접근으로 리스크 관리"

점진적 개선 및 투명한 호환성 등급 제시를 통한 신뢰 확보