



SPRING CAMP

레일웨이 지향 프로그래밍과 Spring

이선협 @kciter

발표자 소개

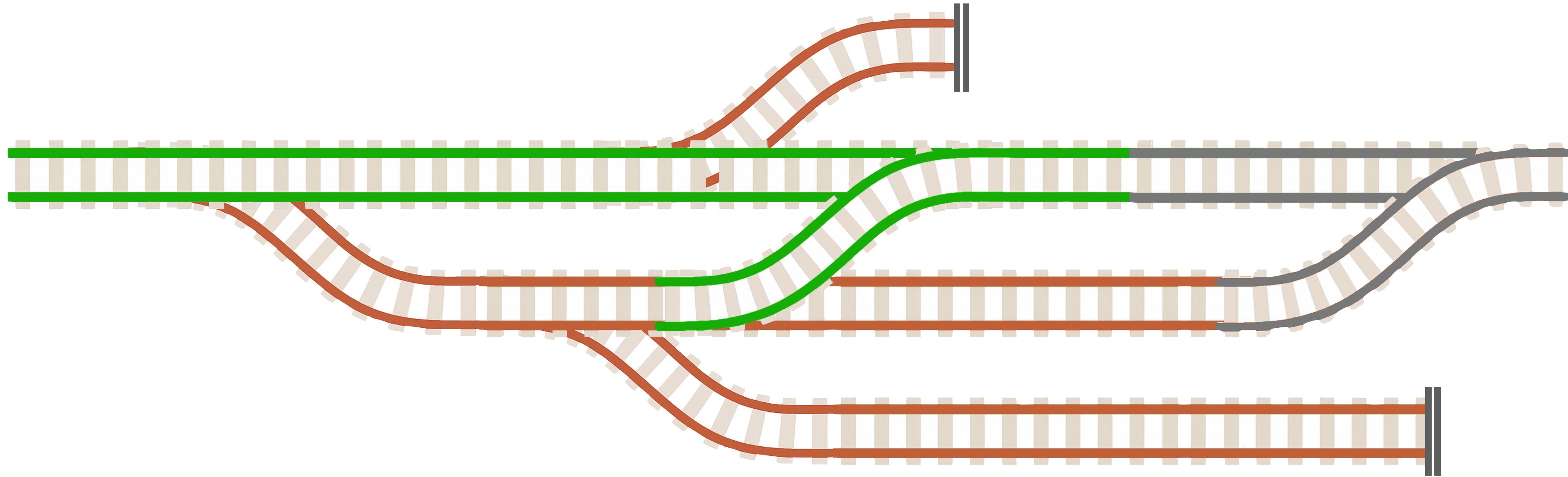
- 이선협(@kciter)
- 12년차 개발자
- 마플코퍼레이션, 플랫폼 엔지니어 리드
- <https://github.com/kciter>
- <https://kciter.so>

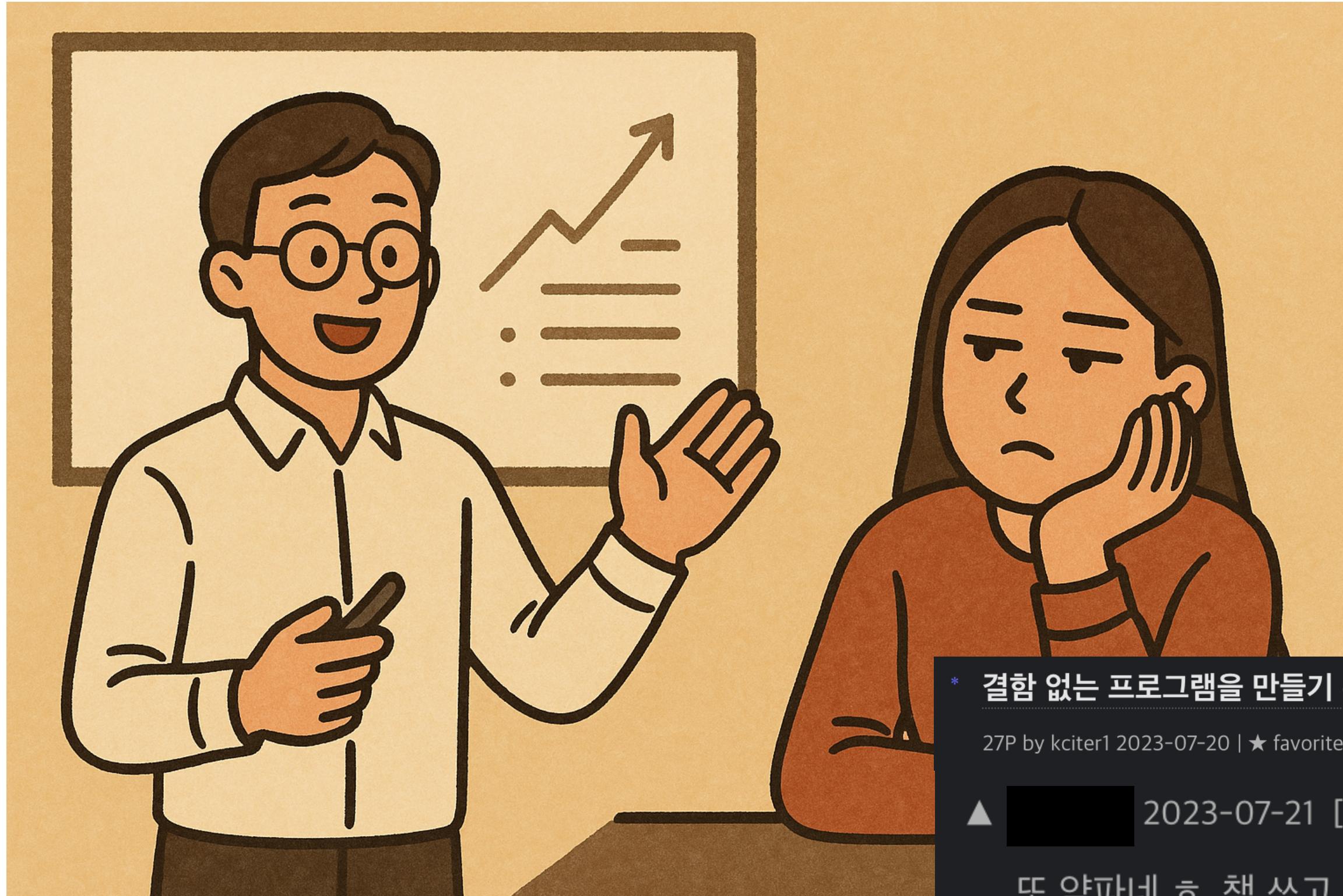


들어가며



살짝 살펴보면 이런 느낌이에요





* 결함 없는 프로그램을 만들기 위한 Railway-Oriented Programming ↗ (kciter.so)

27P by kciter1 2023-07-20 | ★ favorite | 댓글 7개

▲ [REDACTED] 2023-07-21 [-]

또 약파네 ㅎ. 책 쓰고 수입 좀 올리려고? 요즘은 잘 안통하는데?

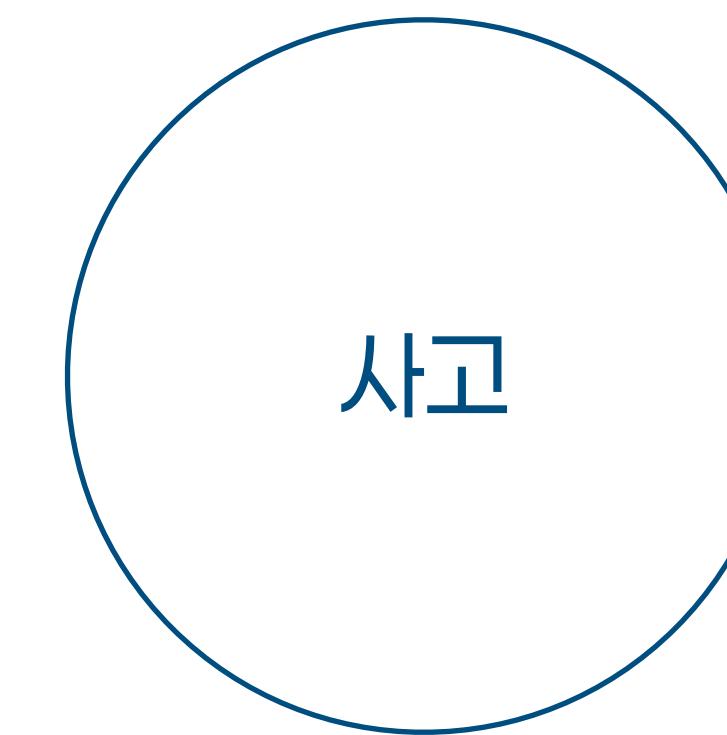
답변달기

방법론은 두 가지 관점에서 살펴봐야 해요

기술

사고

방법론은 두 가지 관점에서 살펴봐야 해요



방법론은 두 가지 관점에서 살펴봐야 해요



레일웨이 지향 프로그래밍

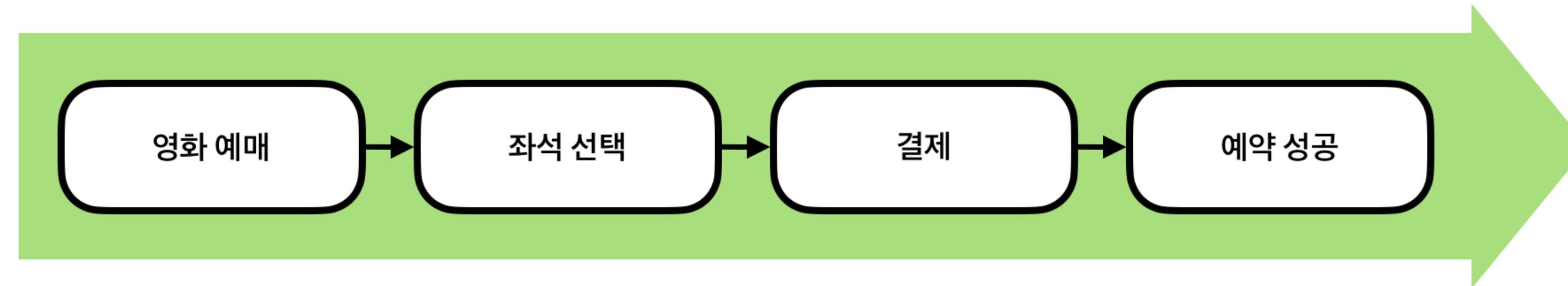
소프트웨어의 목적은?

사용자에게 원하는 결과를 제공하는 것

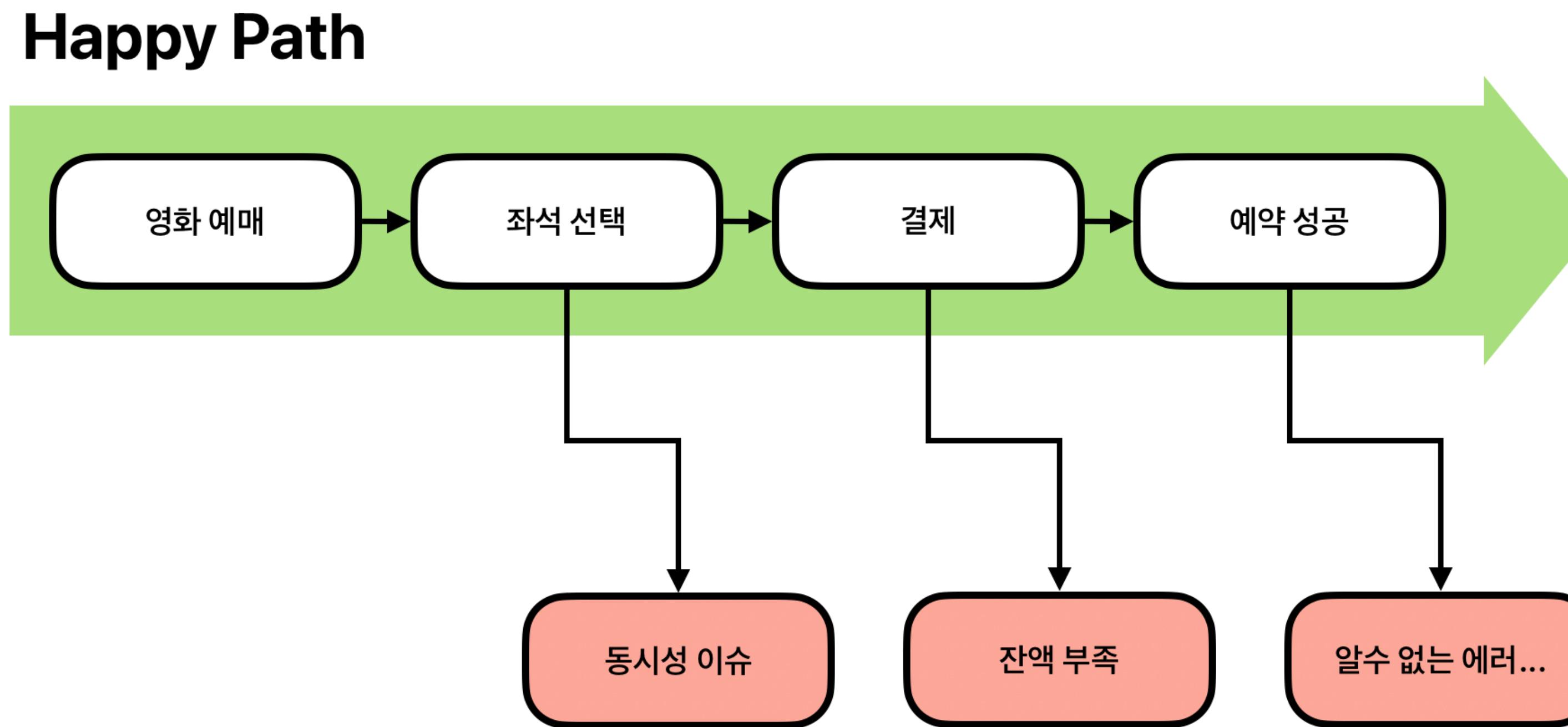
(~~그래야 돈을 벌 수 있음...~~)

우리가 기대하는 소프트웨어의 모습

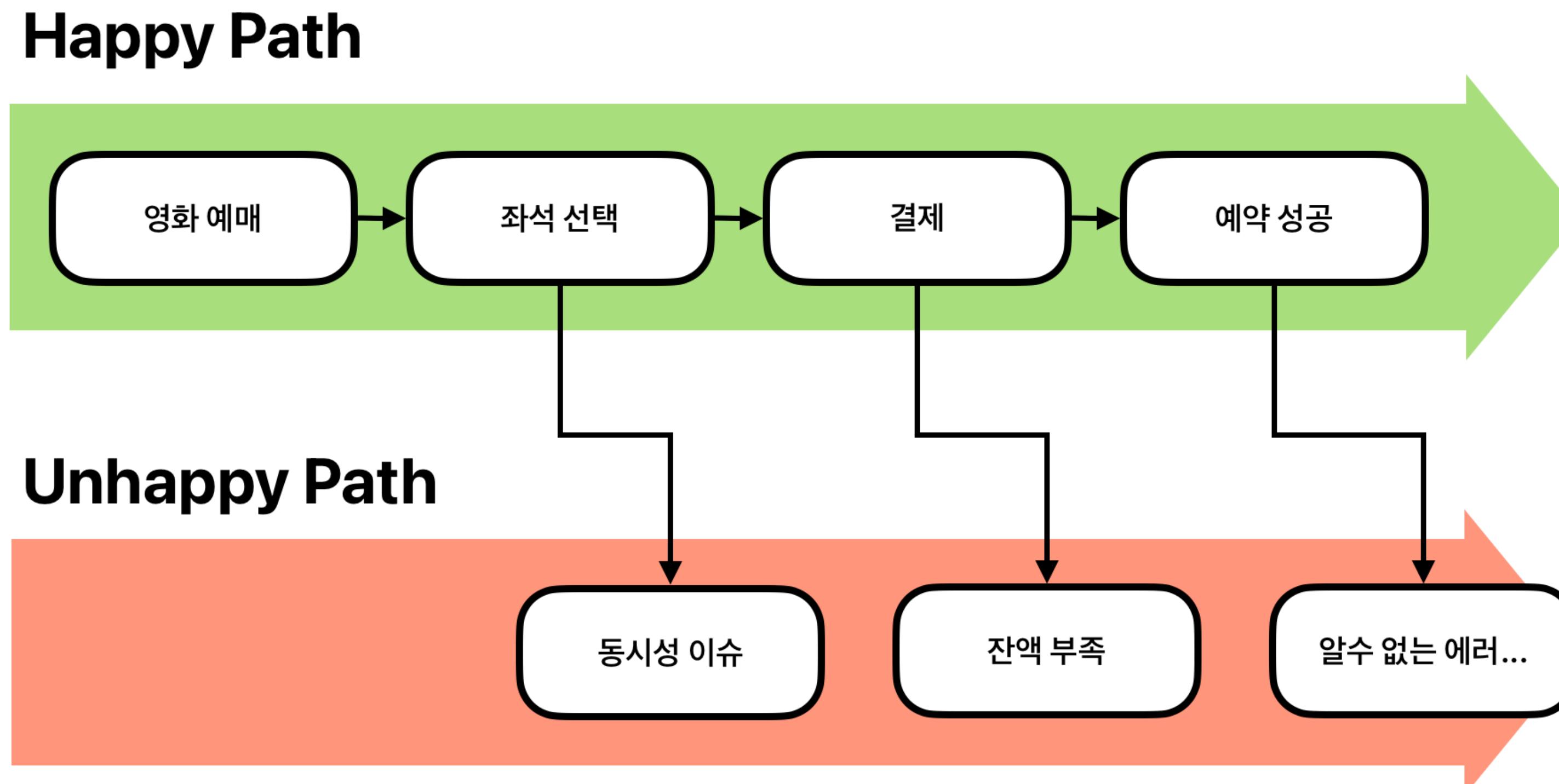
Happy Path



그러나 현실은...



그러나 현실은...



현실적으로 예외를 피할 수는 없어요

- 데이터베이스 접근
- 외부 API 호출
- 비즈니스 예외
- ...

이런 종류의 예외는 **복구**나 **분기 처리**가 필요할 수 있어요

예외를 잘 처리하는 것이 잘만든 소프트웨어의 핵심

얼마나 우아하게 예외를 처리하는지가 소프트웨어의 안정성과 생산성에 영향을 줘요

소프트웨어를 조금 더 단순하게 추상화 해볼까요?

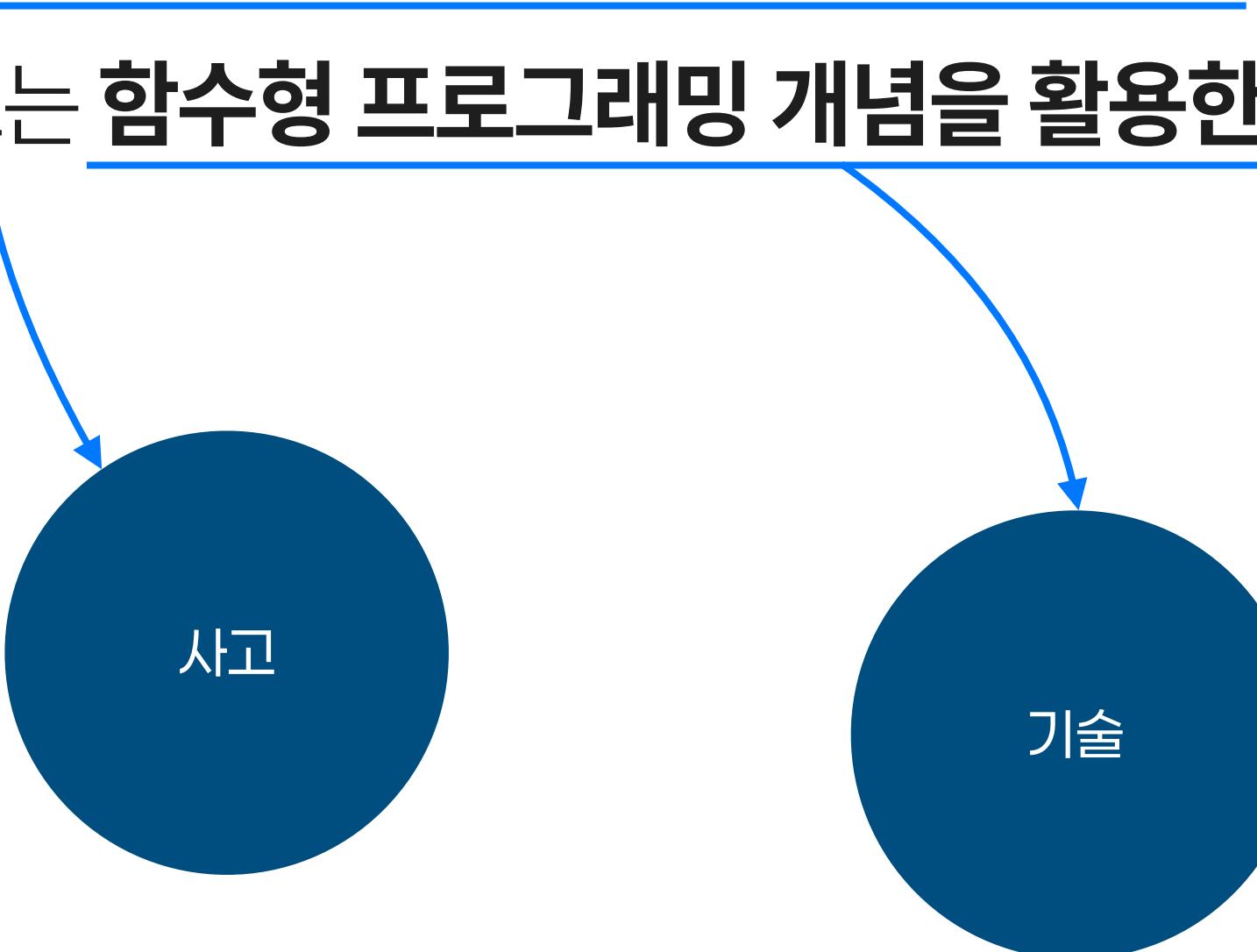
- 소프트웨어의 동작을 성공과 실패로 나눠봅시다
- 실패는 Happy Path를 벗어난 에러 혹은 예외 상황
- 에러, 예외를 다루는 것은 흐름 제어를 하는 것
- 이 두 개념을 합친다면?

Railway-Oriented Programming

- 2014년, Scott Wlashin이 처음 소개했어요
- 레일웨이 지향 프로그래밍은 로직의 흐름을 선로(Railway)로 추상화하는 방법론이에요
- **성공적인 흐름에 대한 선로와 실패에 대한 선로로 나누어요**
- 구체적으로는 **함수형 프로그래밍 개념을 활용한 예외 처리 방법**이라 볼 수 있어요

Railway-Oriented Programming

- 2014년, Scott Wlashin이 처음 소개했어요
- 레일웨이 지향 프로그래밍은 로직의 흐름을 선로(Railway)로 추상화하는 방법론이에요
- **성공적인 흐름에 대한 선로와 실패에 대한 선로**로 나누어요
- 구체적으로는 **함수형 프로그래밍 개념을 활용한 예외 처리 방법**이라 볼 수 있어요

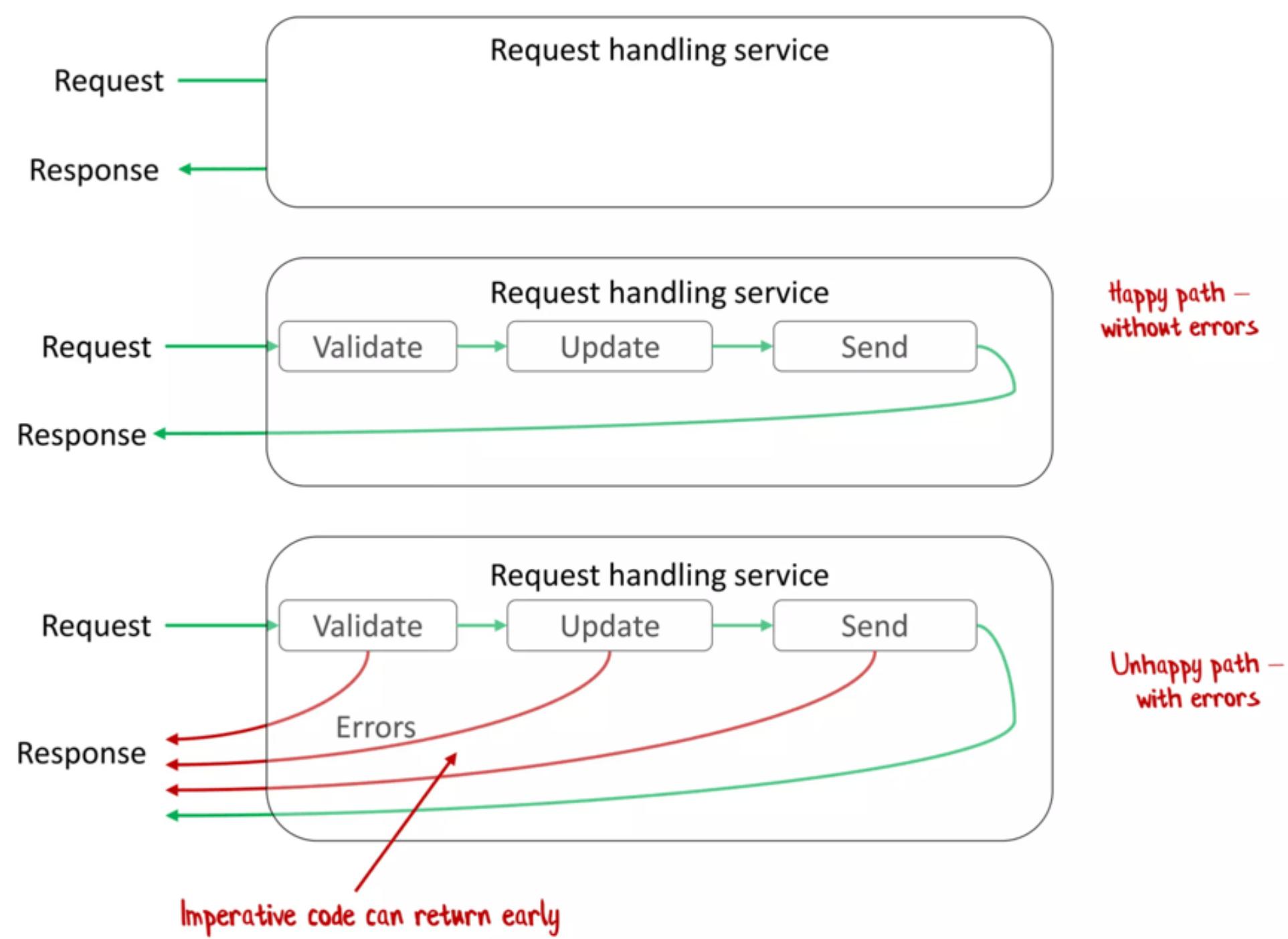


함수형적인 예외 처리

우리가 잘 알고 있는 예외 처리

- 보통 throw와 try-catch를 사용해요
- 예외가 발생하면 **함수를 종료하고 호출한 곳으로 예외**를 던져요
- 이러한 방식은 극단적으로는 Goto 문을 사용한 것과 유사해요

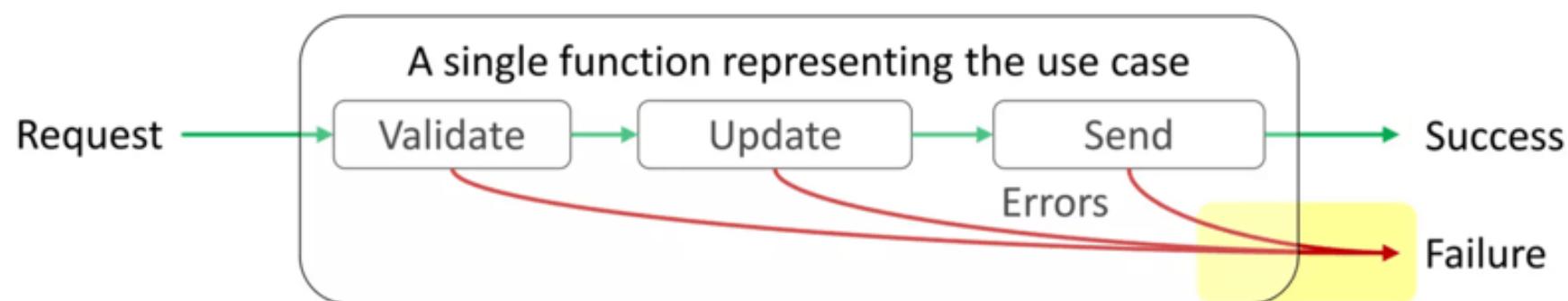
Request/response (non-functional) design



함수형적인 예외 처리

- 함수형적인 방식은 **예외를 값으로** 취급해요
- 예외가 발생해도 중단하지 않고 다음 단계로 값을 넘겨요
- 최종적으로 로직의 마지막 단계에서 **성공인지 실패인지 타입**으로 나타낼 수 있어요

Functional design

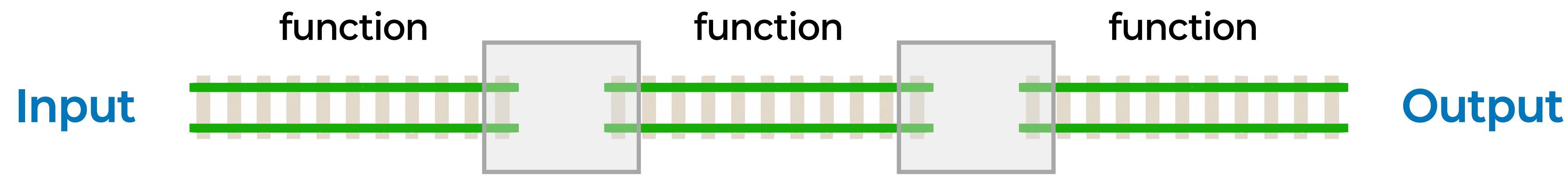


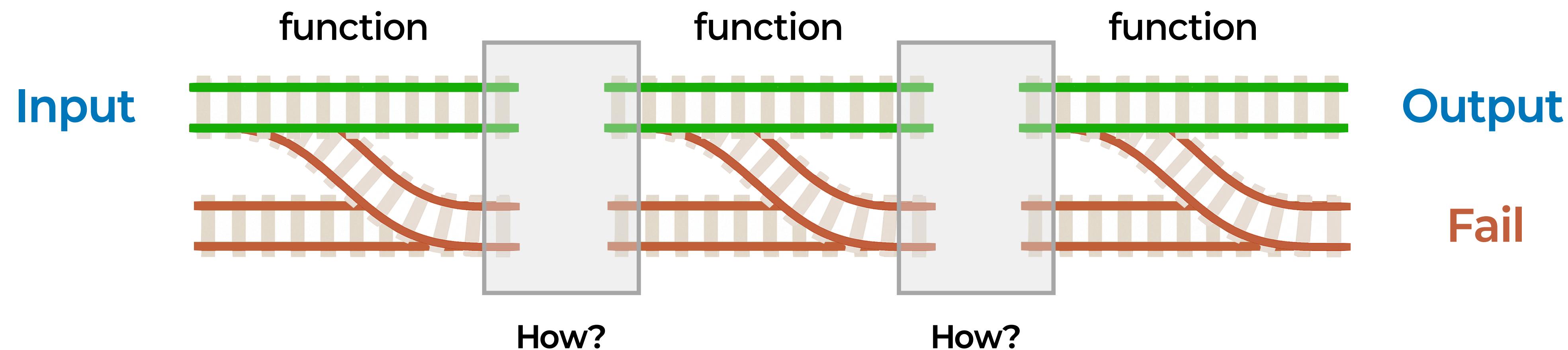
How can a function have more than one output?

```
type Result =  
| Success  
| Failure
```

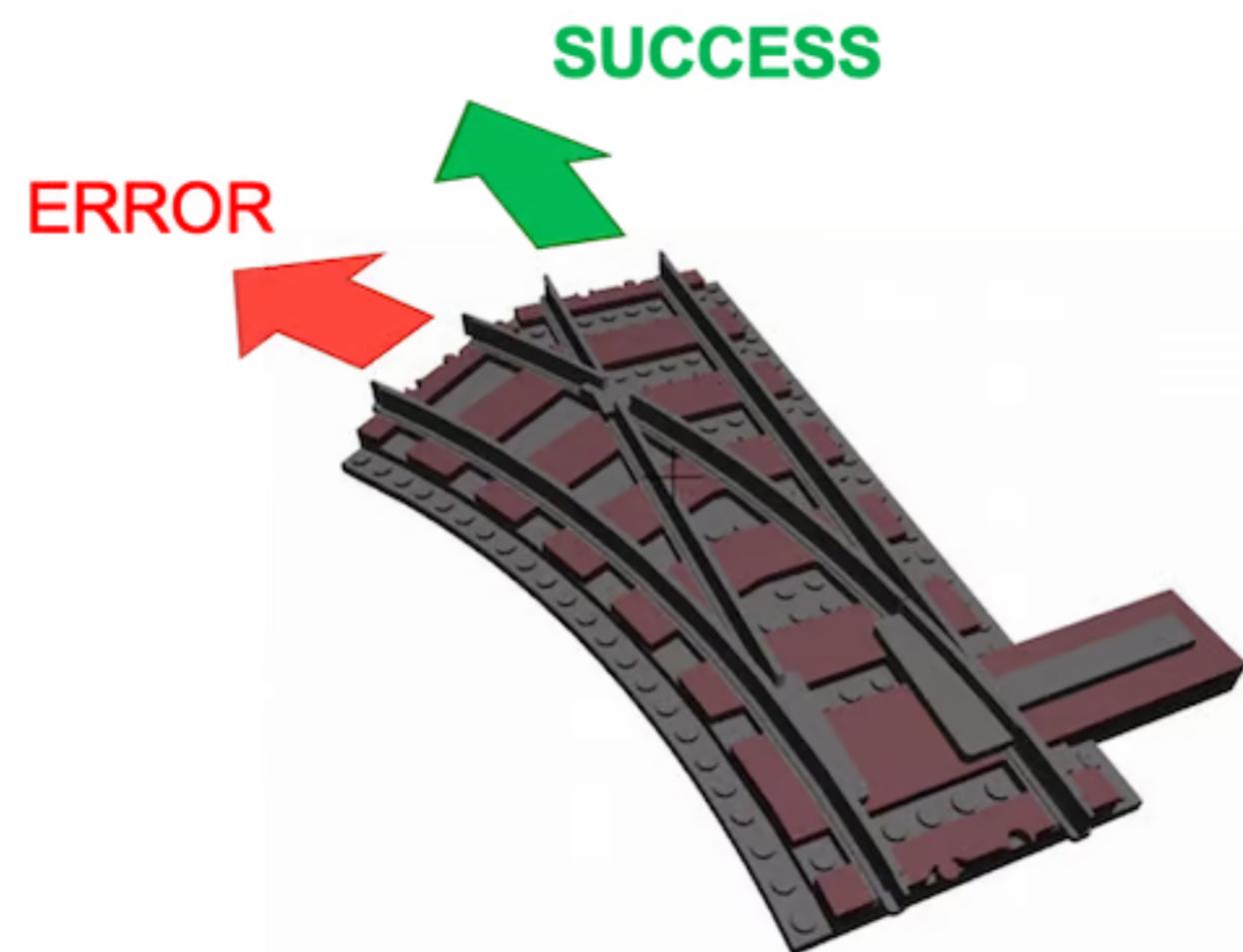
Much more generic – but no data!







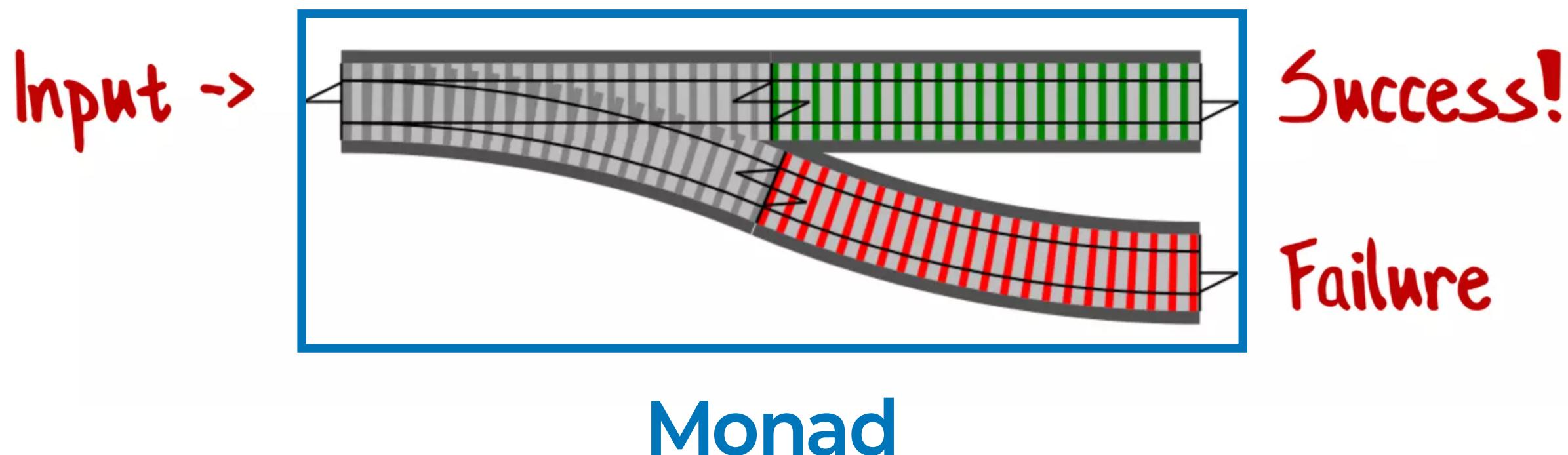
선로(성공과 실패)를 어떻게 코드로 표현할까요?



모나드 (Monad)

- 성공과 실패를 타입으로 표현할 수 있어요
- 모나드는 합성이 가능해서 여러 연산을 순차적으로 연결할 수 있어요
- 값을 꺼내는 시점에 성공과 실패를 확정할 수 있어요

즉, 선로로 표현하기 적합해요



모나드, 알아야만 쓸 수 있을까요?

그렇지 않습니다!

모나드, 이것만 알아도 돼요

- 모나드는 **값을 감싼 컨테이너**예요
- 모나드는 **합성**이 가능해요
- **flatMap**을 써본 적이 있다면 충분해요!

```
listOf(1, 2, 3)
    .flatMap {
        // 각 요소를 두 개의 요소로 변환
        listOf(it, it * 2)
    }

// 결과: [1, 2, 2, 4, 3, 6]
```

Kotlin에서 ROP 시작하기

- 타입 시스템이 강력하고 함수형적인 프로그래밍이 가능해요
- Kotlin에서 사라진 Checked Exception을 대신하기 좋아요

Kotlin 기본 Result 사용하기

Result.success와 Result.failure를 사용하여 성공과 실패를 처리할 수 있어요

```
fun getArraySize(list: List<Int>?): Result<Int> {
    return if (list != null) {
        Result.success(list.size) // 성공
    } else {
        Result.failure(IllegalArgumentException("list must not be null")) // 실패
    }
}
```

혹은 runCatching 함수를 이용해서 예외를 Result로 변환할 수도 있어요

```
fun getArraySize(list: List<Int>?): Result<Int> {
    return runCatching {
        if (list == null) {
            throw IllegalArgumentException("list must not be null") // 실패
        }
        list.size // 성공
    }
}
```

ROP를 할 때 기본 제공 Result는 권장하지 않아요

- 예외에 대한 타입 정보를 표현할 수 없어요
- flatMap을 이용한 합성이 불가능해요
- 편의를 위한 확장 함수가 부족해요
- 성공과 실패에 대한 패턴 매칭이 불가능해요

ROP를 할 때 기본 제공 Result는 권장하지 않아요

- 예외에 대한 타입 정보를 표현할 수 없어요
- flatMap을 이용한 합성이 불가능해요
- 편의를 위한 확장 함수가 부족해요
- 성공과 실패에 대한 패턴 매칭이 불가능해요

✨ 직접 만들어서 해결할 수 있어요!

저는 Effect 모나드를 직접 만들었어요

- <https://gist.github.com/kciter/9dc9ffbf80d1ef146a556fef66a38840>
- flatMap을 통한 합성이 가능해요
- 수신 객체를 이용한 Monad Comprehension이 가능해요
- recover, unwrap, zip, fold 등 유용한 확장 함수를 제공해요
- Effect로 이름 지은 이유는 기본 제공 Result와 구분하기 위해서에요
- 만약 직접 만드는 것이 싫다면...
 - kotlin-result라는 라이브러리를 사용할 수도 있어요
 - 1.x와 2.x 버전으로 나뉘어요
 - 2.x 버전은 value class와 inline fun을 이용하여 성능이 향상됐지만 컴파일 타임 안정성이 떨어져요

```
sealed class Effect<out V, out E: Throwable> {
    class Success<out V>(val value: V): Effect<V, Nothing>() {
        // ...
    }

    class Failure<out E: Throwable>(val error: E): Effect<Nothing, E>() {
        // ...
    }
}
```

한 가지 예제를 살펴볼까요?

Kotlin에서 기존 try-catch 방식으로 예외 처리를 한다면 다음과 같이 작성할 수 있어요

```
fun validate(request: RegisterUserRequest): Unit
fun createUser(request: RegisterUserRequest): User
fun sendCreateUserNotificationEmail(createdUser: User): Unit

fun registerUser(request: RegisterUserRequest): User {
    try {
        validateCreateUserRequest(request)
        val createdUser = createUser(validatedRequest)
        sendUpdateUserNotificationEmail(it, createdUser)
        return createdUser
    } catch (e: Exception) {
        // 어떤 예외가 발생했는지 알기 어려워요
        // 예외 처리 전략을 세우기 위해서는 각 함수의 내용을 살펴봐야 해요
        throw e
    }
}
```

모나드 방식의 예외 처리

만약 Effect 모나드를 사용한다면 다음과 같이 작성할 수 있어요

```
// sealed class를 사용하여 합타입으로 예외를 표현할 수 있어요
sealed class UserException: Exception()
data class ValidationException(override val message: String): UserException()
data class FailedToCreateUserException(override val message: String): UserException()
data class EmailSendingException(override val message: String): UserException()

fun validate(request: RegisterUserRequest): Effect<Unit, ValidationException>
fun createUser(request: RegisterUserRequest): Effect<User, FailedToCreateUserException>
fun sendCreateUserNotificationEmail(createdUser: User): Effect<Unit, EmailSendingException>

fun registerUser(request: RegisterUserRequest): User {
    // 명시적으로 에러 타입을 알 수 있어요
    val result: Effect<User, UserException> =
        validate(request)
            .flatMap { createUser(request) }
            .flatMap { sendCreateUserNotificationEmail(it) }

    // ...
}
```

모나드 방식의 예외 처리

만약 Effect 모나드를 사용한다면 다음과 같이 작성할 수 있어요

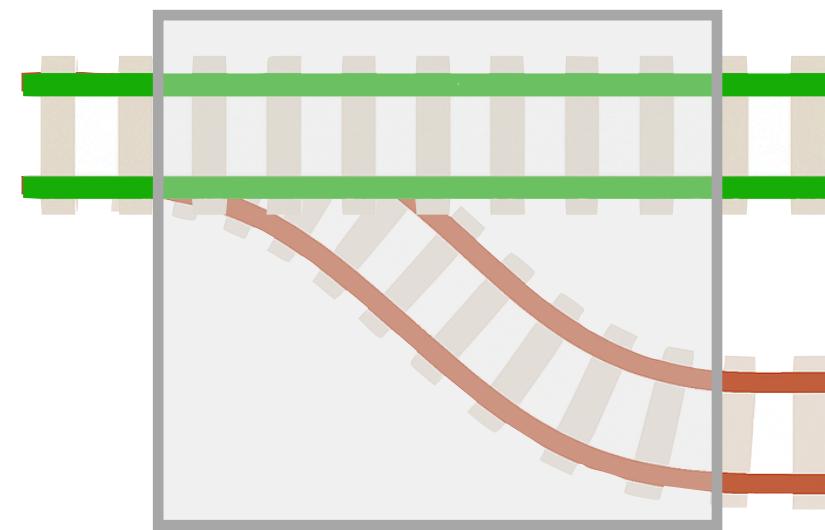
```
// sealed class를 사용하여 합타입으로 예외를 표현할 수 있어요
sealed class UserException: Exception()
data class ValidationException(override val message: String): UserException()
data class FailedToCreateUserException(override val message: String): UserException()
data class EmailSendingException(override val message: String): UserException()

fun validate(request: RegisterUserRequest): Effect<Unit, ValidationException>
fun createUser(request: RegisterUserRequest): Effect<User, FailedToCreateUserException>
fun sendCreateUserNotificationEmail(createdUser: User): Effect<Unit, EmailSendingException>

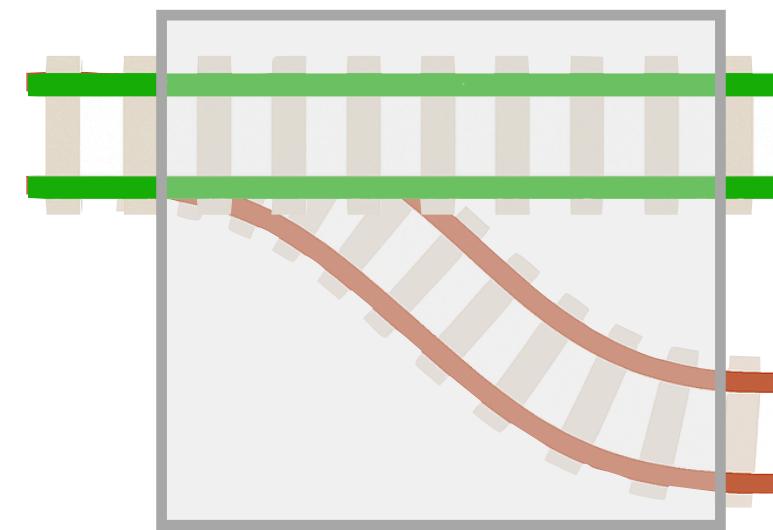
fun registerUser(request: RegisterUserRequest): User {
    // 명시적으로 에러 타입을 알 수 있어요
    val result: Effect<User, UserException> =
        validate(request)
            .flatMap { createUser(request) } → 합성!
            .flatMap { sendCreateUserNotificationEmail(it) }

    // ...
}
```

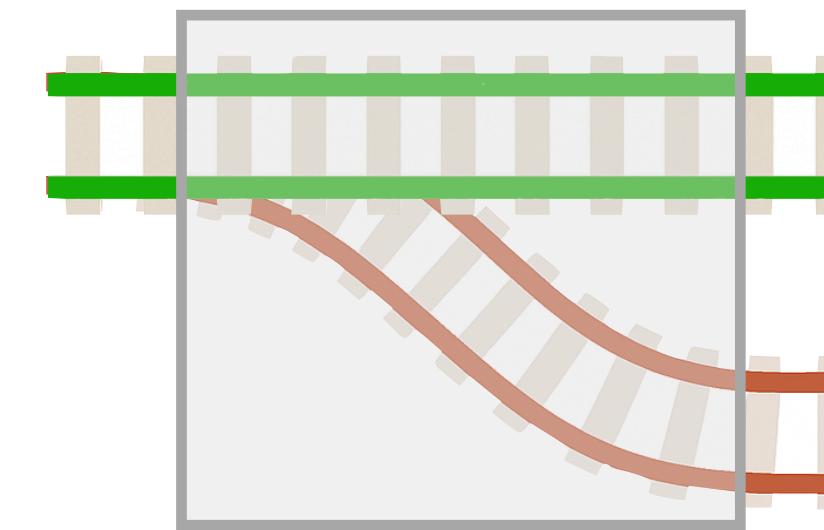
validate

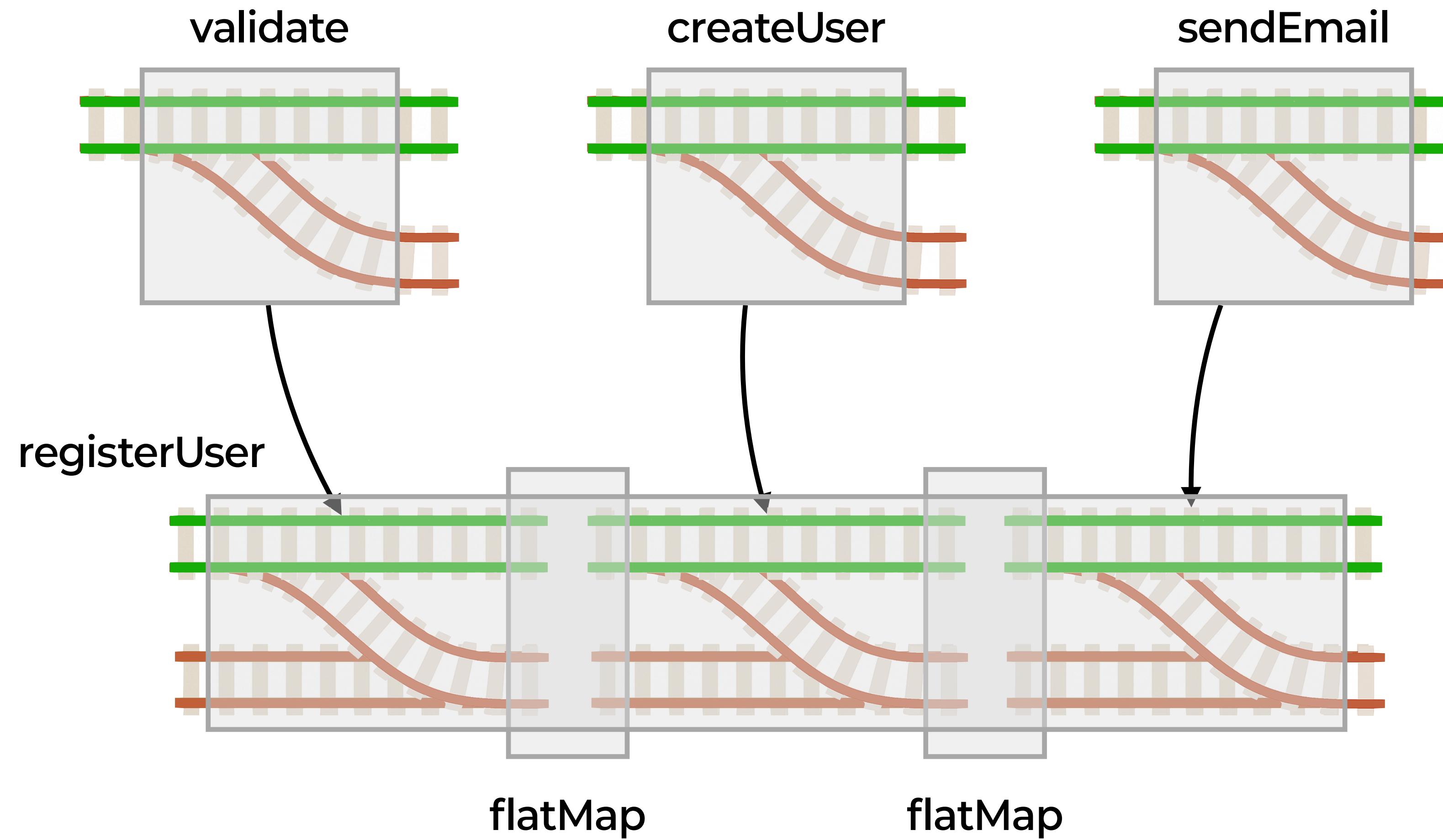


createUser



sendEmail



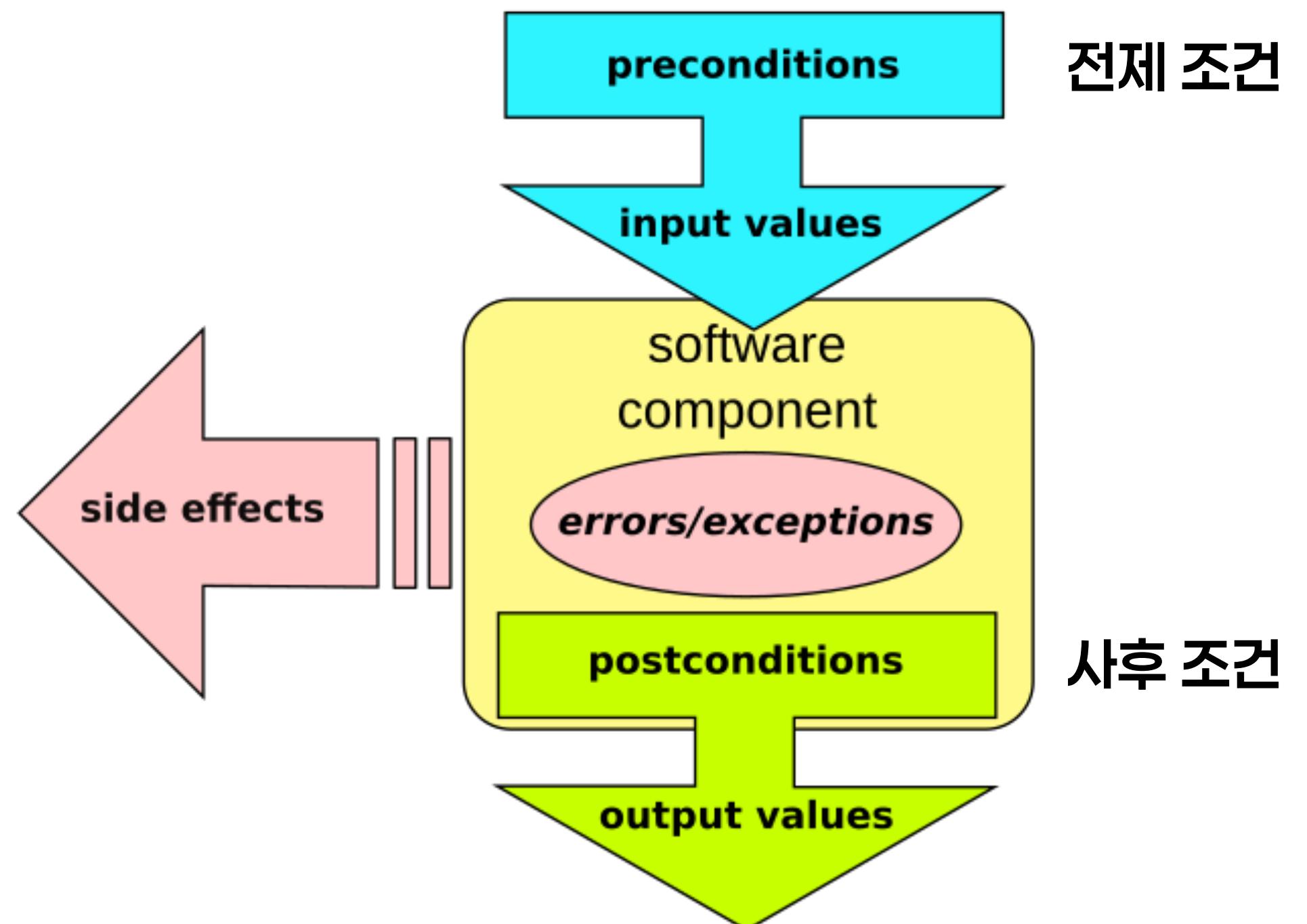


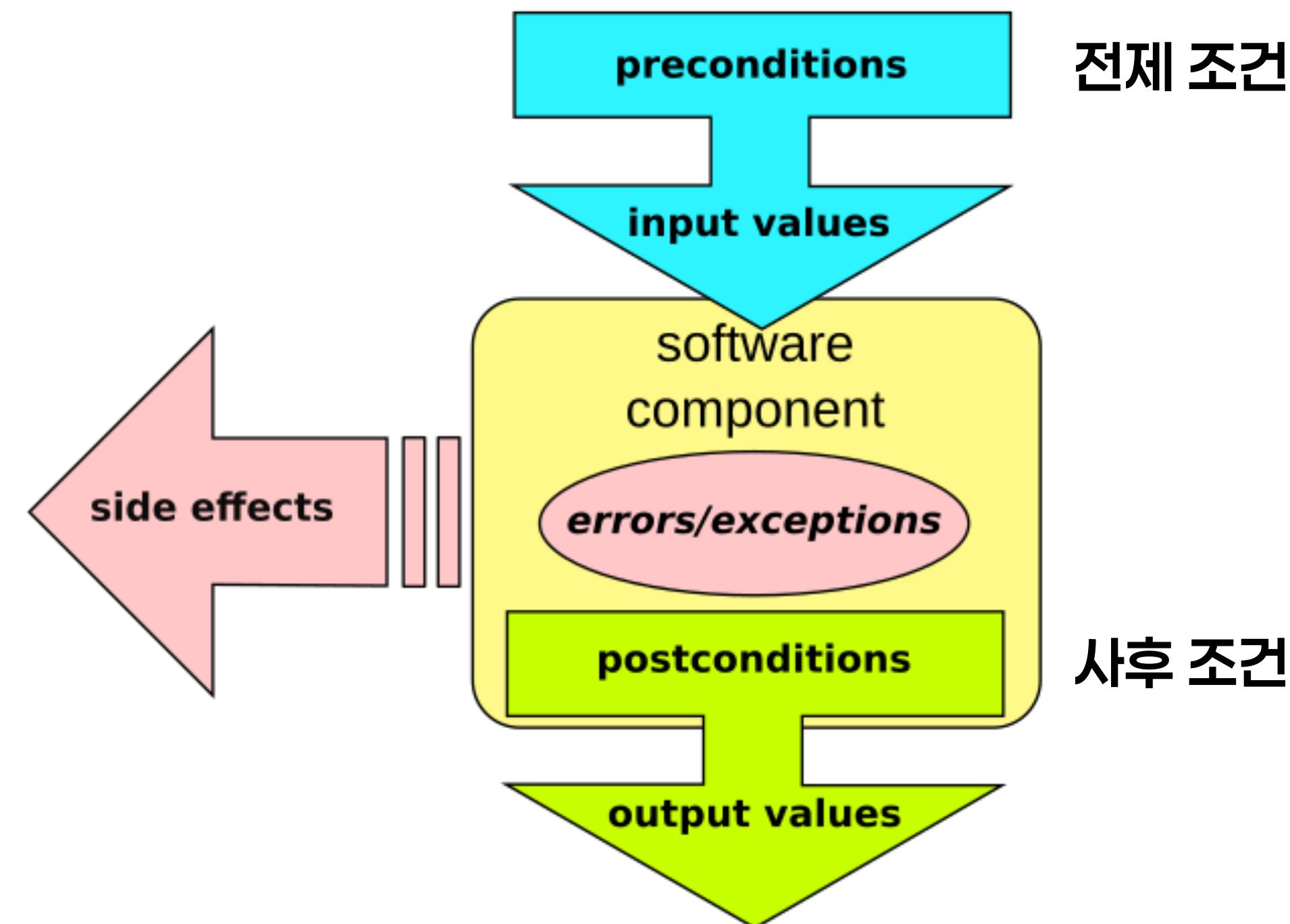
모나드와 합성을 통해 성공과 실패로 소프트웨어를 구성할 수 있어요

계약에 의한 설계

계약에 의한 설계이란?

- 런타임 시점에 **제약과 조건을 명확하게 표현하기 위한 설계 방법**이에요
- 로직을 성공과 실패로 구분하는 ROP와 궁합이 좋아요





소프트웨어의 구성 요소(함수, 객체 등)는 **계약**을 준수할 의무가 있어요

전제 조건

- 입력 값에 대한 계약을 검증하는 로직을 말해요
- LBYL(Look Before You Leaf)와 유사해요
 - 실패를 미리 검사하고 처리하는 방법이에요
- Guard Clause 패턴이라고도 볼 수 있어요

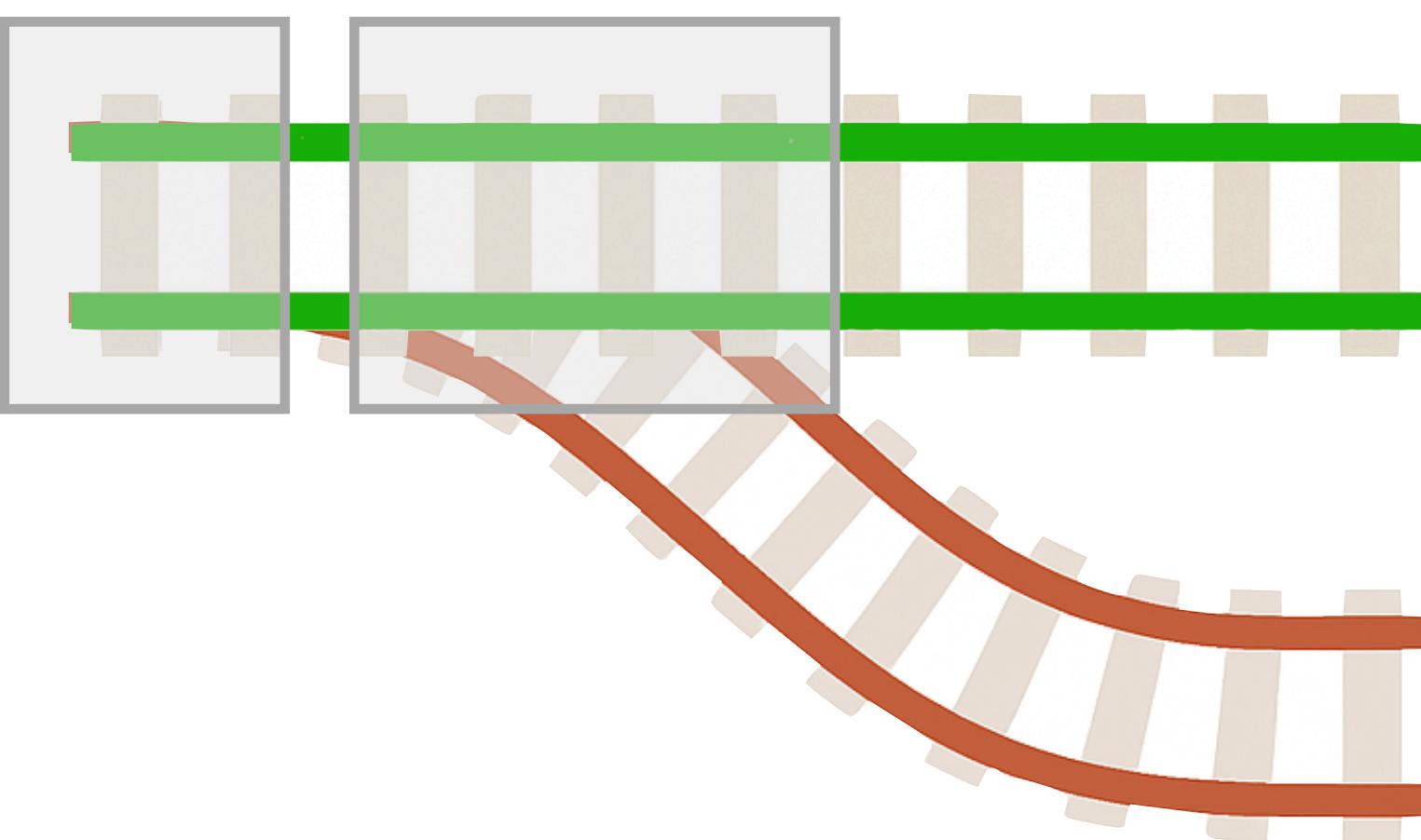
```
fun calculateDiscount(price: Double): Effect<Double, Exception> {  
    if (price < 0) {  
        return Effect.Failure(IllegalArgumentException("가격은 음수가 될 수 없습니다."))  
    }  
    return Effect.Success(price * 0.9) // 10% 할인  
}
```

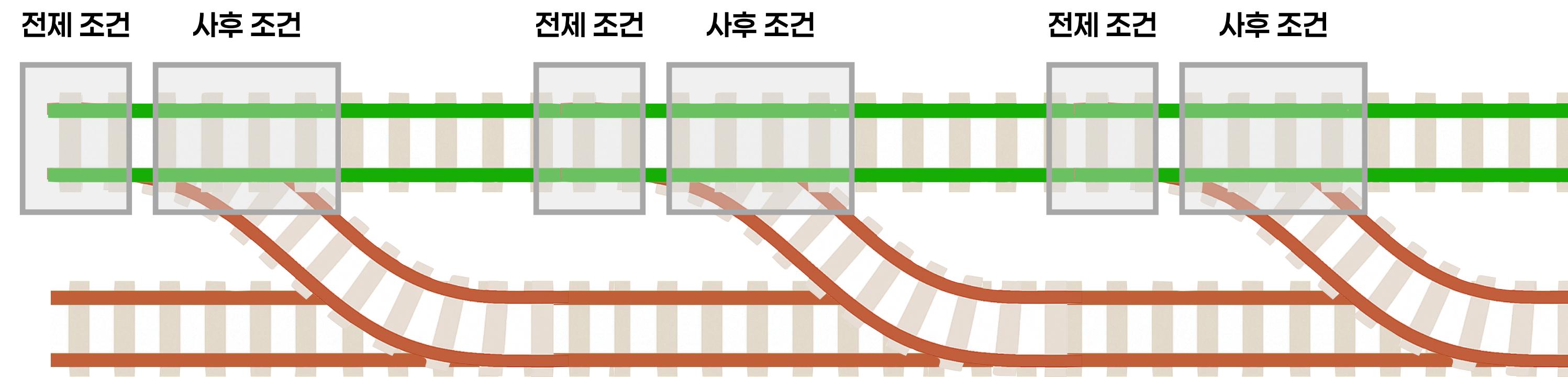
사후 조건

- 응답에 대한 계약을 지키는지 검증하는 로직이에요
- 처리에 실패했을 수도 있고 응답이 원하지 않는 값일 수도 있어요
- EAFP (Easier to Ask Forgiveness than Permission)와 유사해요
 - 일단 실행하고 용서를 구해요

```
fun findUserById(userId: Int): Effect<User, Throwable> {
    return effect { repository.findUserById(userId) }
        .map { user ->
            user ?: throw NotFoundException("User not found")
        }
}
```

전제 조건 사후 조건





각 선로는 계약을 이행하는 함수라고 할 수 있어요

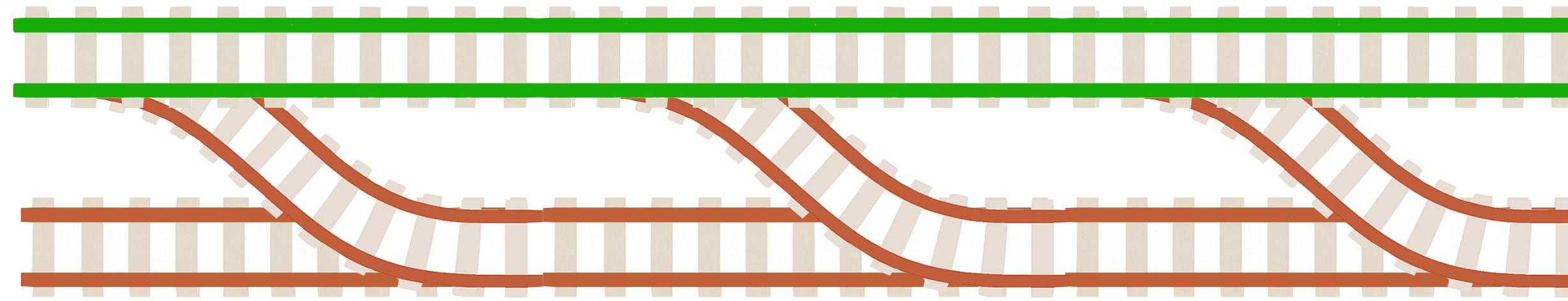




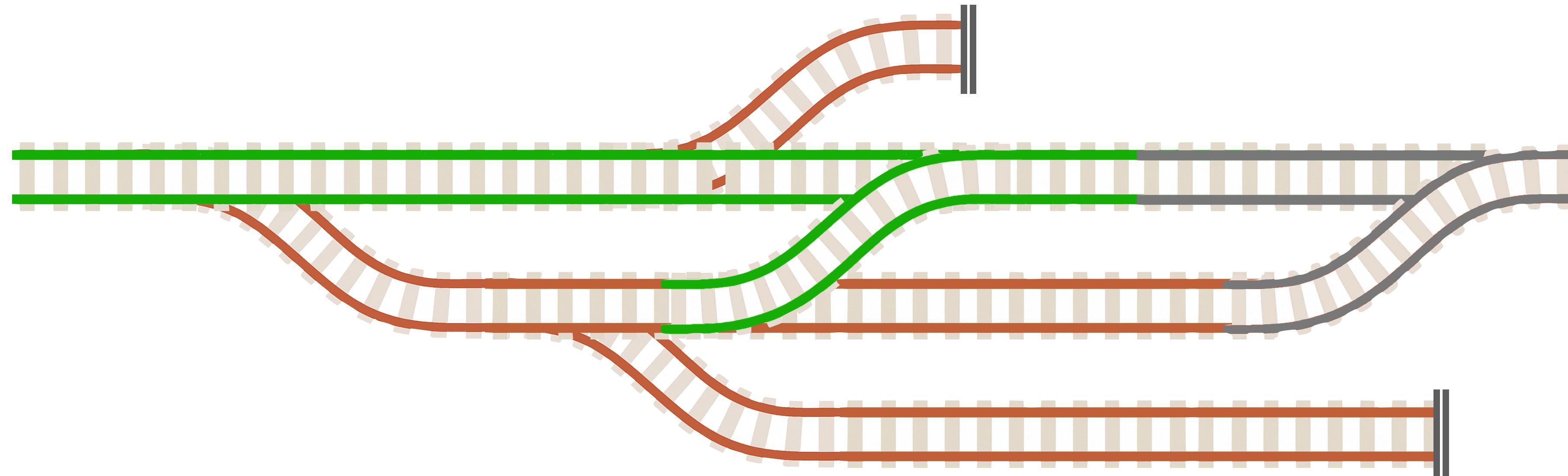
타입은 함수와 데이터가 준비해야 하는 엄격한 계약!

+ 타입을 통해 가독성을 올리는 것도 가능해요

```
sealed class UserException: Exception()
data class ValidationException(override val message: String): UserException()
data class FailedToCreateUserException(override val message: String): UserException()
data class EmailSendingException(override val message: String): UserException()
```

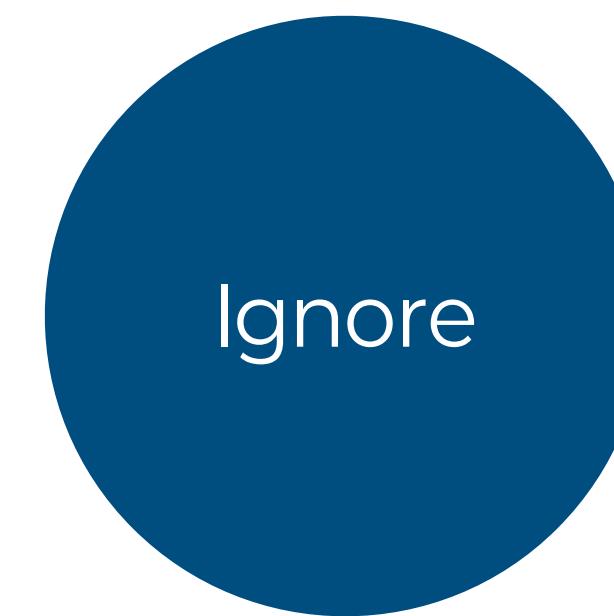
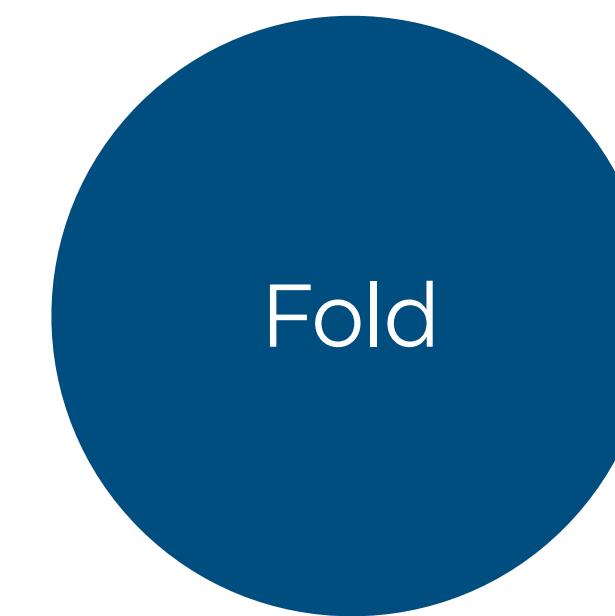


이렇게 우리는 안전한 소프트웨어를 만들 수 있게 됐습니다?



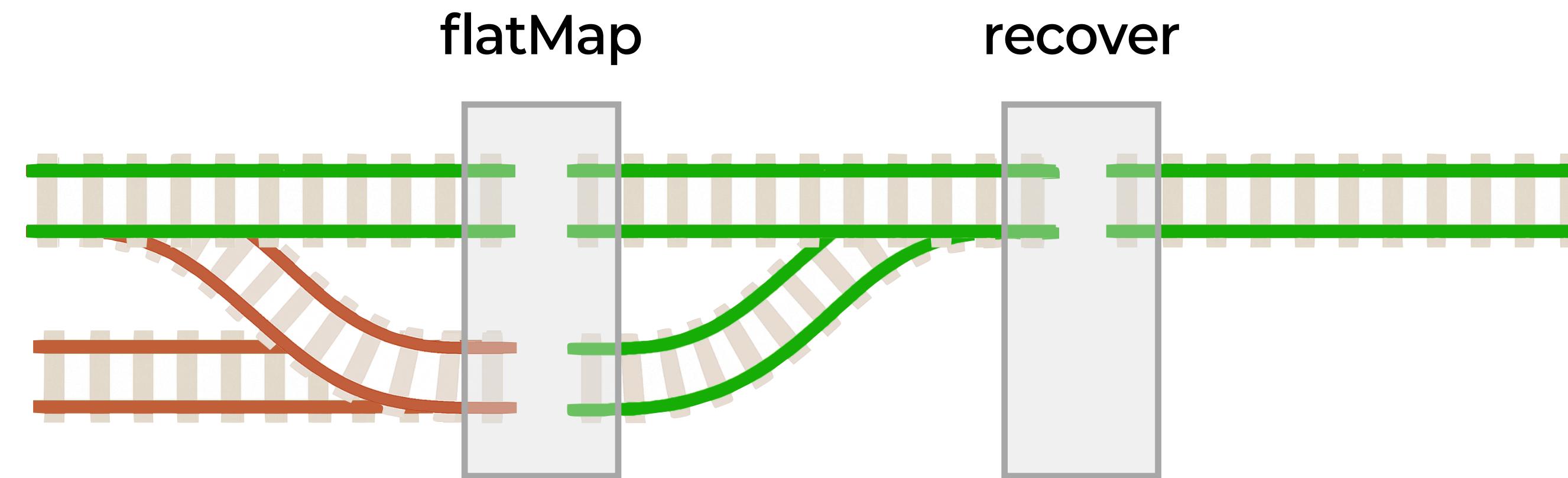
안타깝게도 실제 소프트웨어 개발은 더 복잡해요

다양한 예외 처리 패턴



Recover

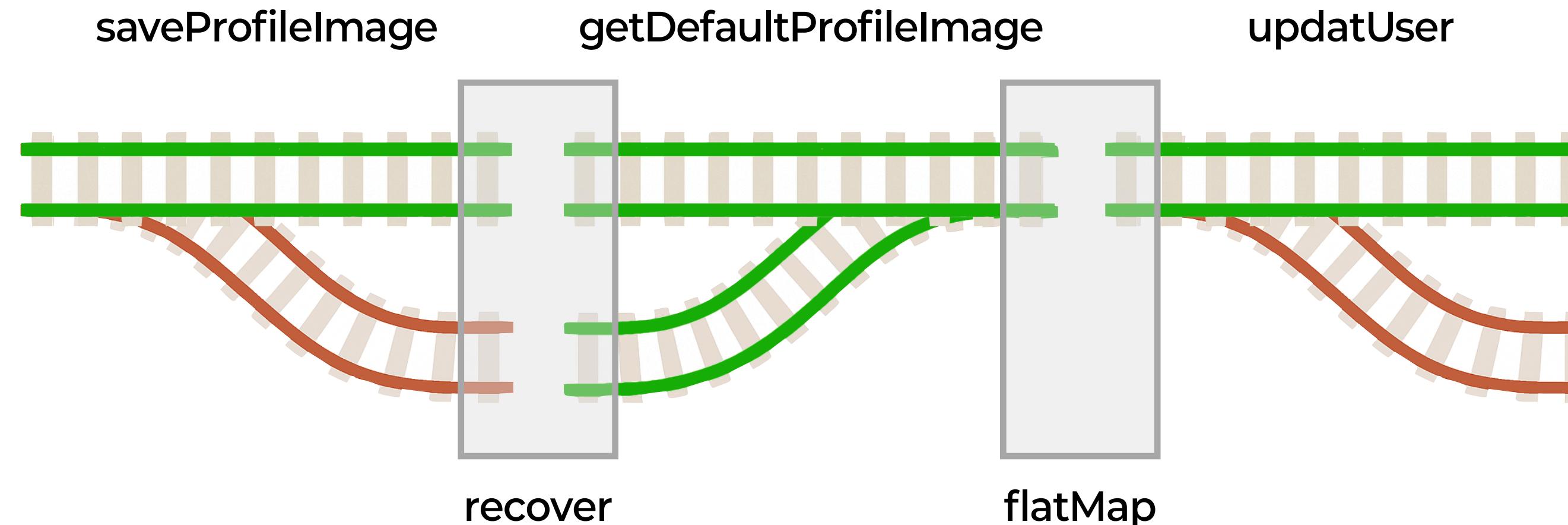
- 예외 중에는 복구가 가능한 경우도 있어요
- recover 함수를 사용하여 복구하면 Effect.Success 타입으로 확정돼요



Recover

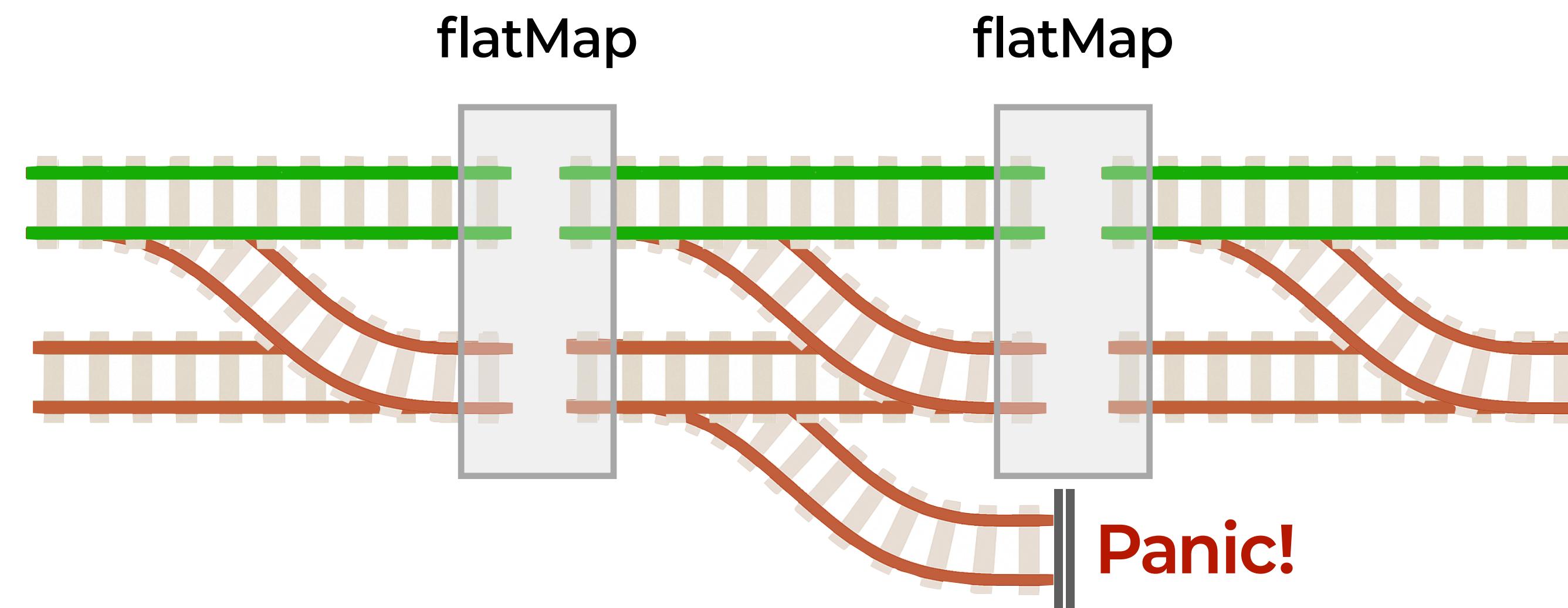
```
fun saveProfileImage(image: ByteArray): Effect<Image, Throwable> = TODO()
fun getDefaultProfileImage(): Image = TODO()
fun updateUser(userId: String, request: UserUpdateRequest): Effect<User, Throwable> = TODO()

fun updateUserProfileImage(userId: String, image: ByteArray): Effect<User, Throwable> {
    return saveProfileImage(image)
        .recover { getDefaultProfileImage() }
        .flatMap { updateUser(userId, UserUpdateRequest(image = it)) }
}
```



Panic

- 복구가 불가능한 예외는 그대로 종료하는 것이 좋아요
- 개발자의 실수나 시스템 에러라고 볼 수 있어요



Panic

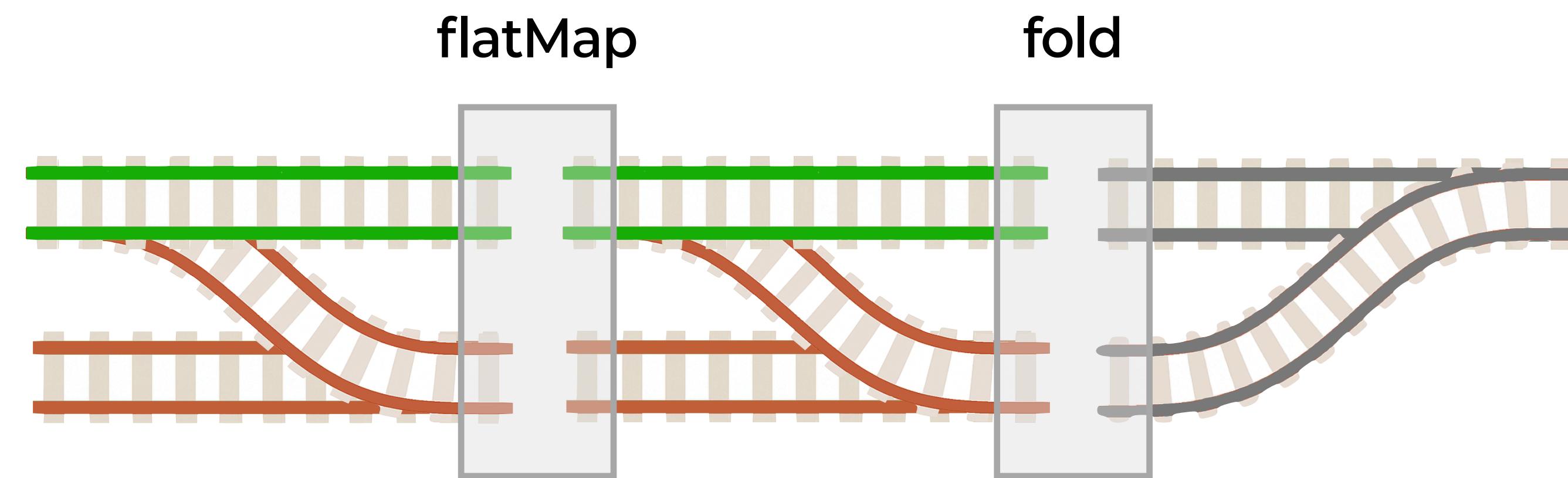
```
fun validate(request: RegisterUserRequest): Effect<Unit, ValidationException> = TODO()
fun createUser(request: RegisterUserRequest): Effect<User, FailedToCreateUserException> = TODO()
fun sendCreateUserNotificationEmail(createdAt: User): Effect<User, EmailSendingException> = TODO()

fun registerUser(request: RegisterUserRequest): User {
    val result = validate(request)
        .flatMap { createUser(request) }
        .flatMap { sendCreateUserNotificationEmail(it) }

    return when (result) {
        is Effect.Success → result.value // 성공한 경우
        is Effect.Failure → {
            when (result.error) {
                is ValidationException → TODO() // 유효성 검사 실패 처리
                is FailedToCreateUserException → TODO() // 사용자 생성 실패 처리
                is EmailSendingException → TODO() // 이메일 전송 실패 처리
                else → { // 알 수 없는 예외는 종료해요
                    logger.error("알 수 없는 예외 발생: ${result.error.message}", result.error)
                    throw result.error
                }
            }
        }
    }
}
```

Fold

- 최종적으로 하나의 값으로 합쳐져야 할 수 있어요
- 보통 Response를 할 때 많이 사용해요

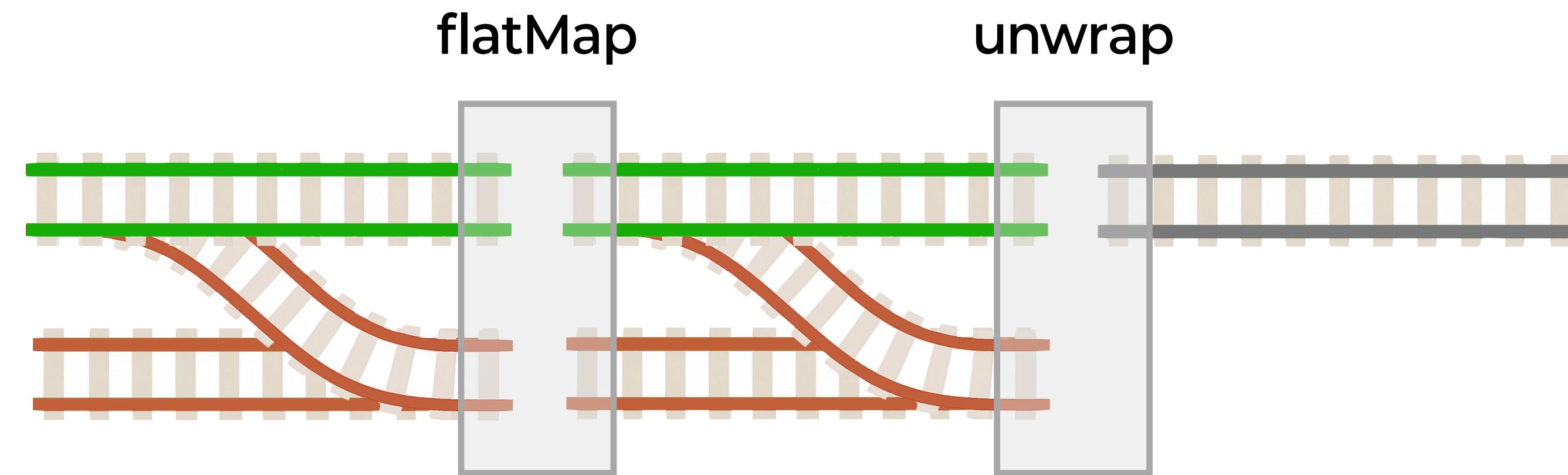


Fold

```
fun registerUser(request: RegisterUserRequest): ResponseEntity<RegisterUserResponse> {
    return validate(request)
        .flatMap { createUser(request) }
        .flatMap { sendCreateUserNotificationEmail(it) }
        .fold(
            onSuccess = {
                ResponseEntity.ok()
                    .body(RegisterUserResponse(id = it.id, name = it.username))
            },
            onFailure = {
                ResponseEntity
                    .status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .build()
            }
        )
}
```

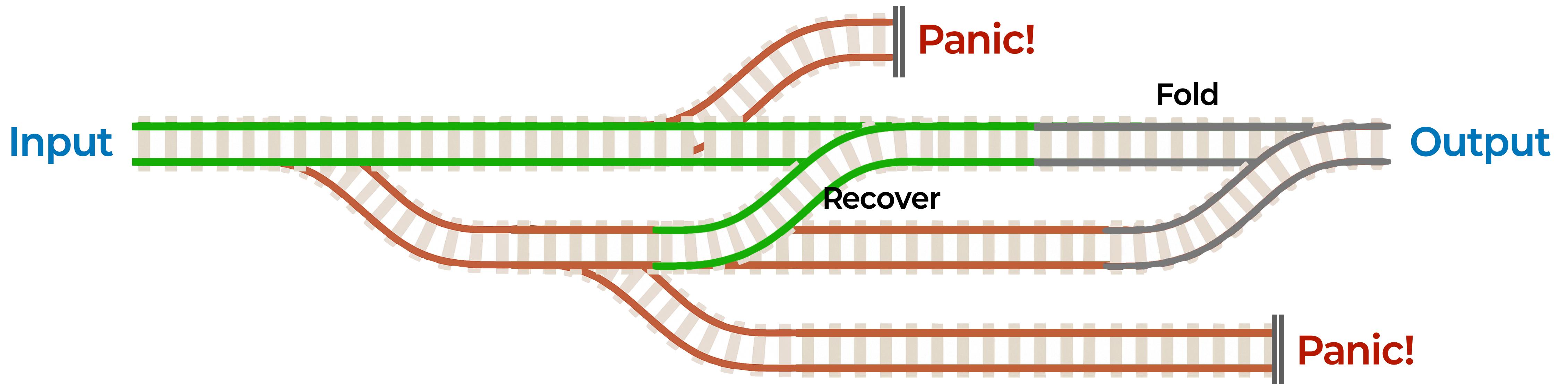
Ignore

- 예외가 있다는 것을 그냥 무시할 수도 있어요
- 절대 예외가 발생하지 않는다는 확신이 있을 때 사용해요
- 만약 예외가 발생하면 그대로 throw를 상위로 던져요



Ignore

```
fun registerUser(request: RegisterUserRequest): User {  
    return validate(request)  
        .flatMap { createUser(request) }  
        .flatMap { sendCreateUserNotificationEmail(it) }  
        .unwrap()  
}
```

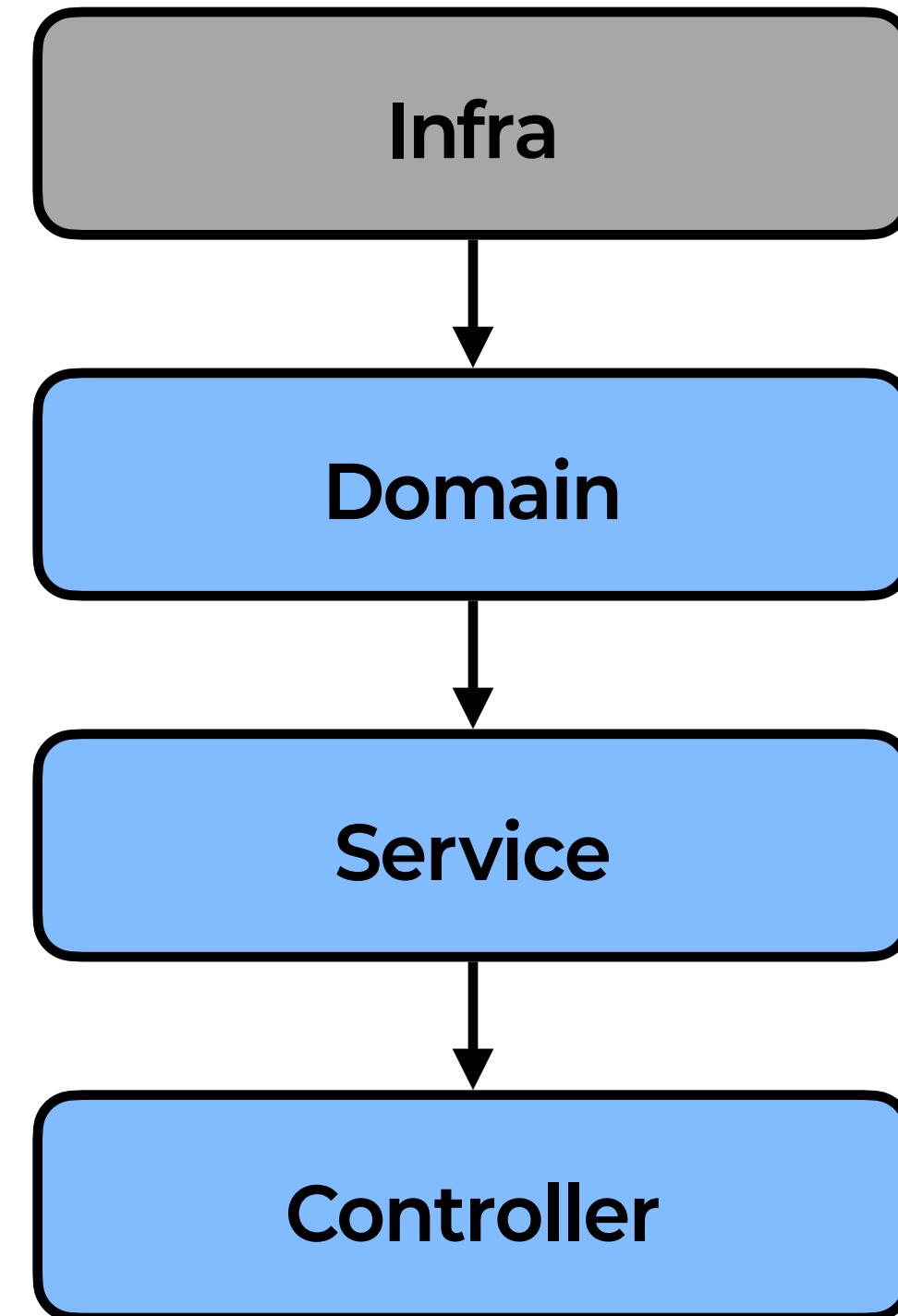


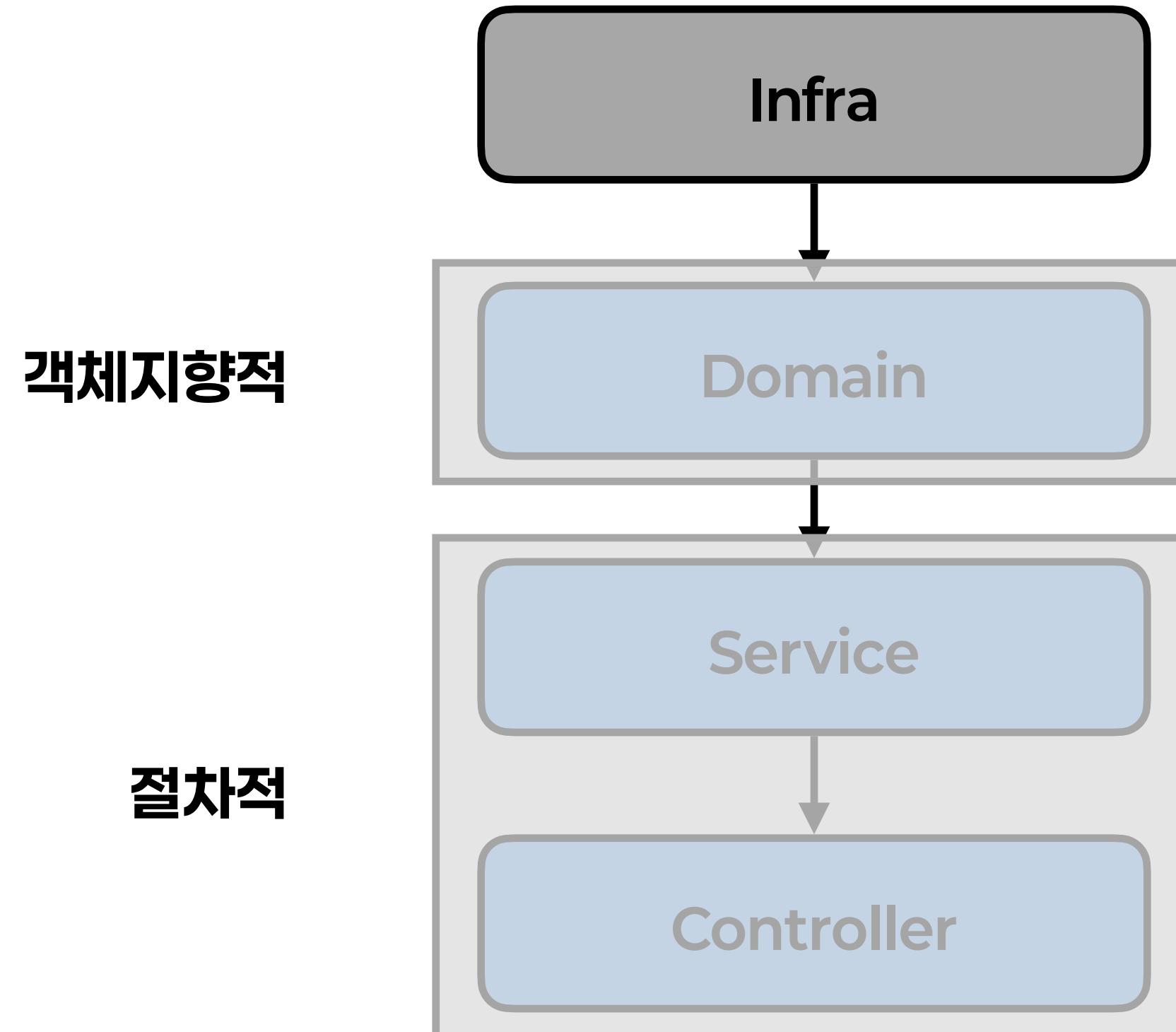
다양한 예외 처리 패턴을 적용하여 로직을 안전하게 **단방향**으로 구현할 수 있어요

정리하면

- 선로를 배치할 때(함수를 만들 때) 계약에 의한 설계를 떠올리면 좋아요
- 타입 시스템을 적극 활용하면 계약에 대한 결과를 처리하기 좋아요
- 예외가 발생했을 때 어떻게 처리할지 전략을 생각하면 좋아요

Spring과 ROP





```
data class User(  
    val id: Int,  
    val username: Name,  
    val email: Email  
) {  
    companion object {  
        fun create(id: Int, username: String, email: String): Effect<User, ValidationException> {  
            return effect {  
                User(id, Name(username), Email(email))  
            }.mapError { ValidationException(it.message) }  
        }  
    }  
  
    fun updateName(newName: String): Effect<User, ValidationException> {  
        return effect {  
            this.copy(username = Name(newName))  
        }.mapError { ValidationException(it.message ?: "Invalid name") }  
    }  
  
    // ...  
}
```

도메인 객체가 항상 **계약**을 준수하도록 **참조 투명성**을 유지하며 코드를 작성할 수 있어요

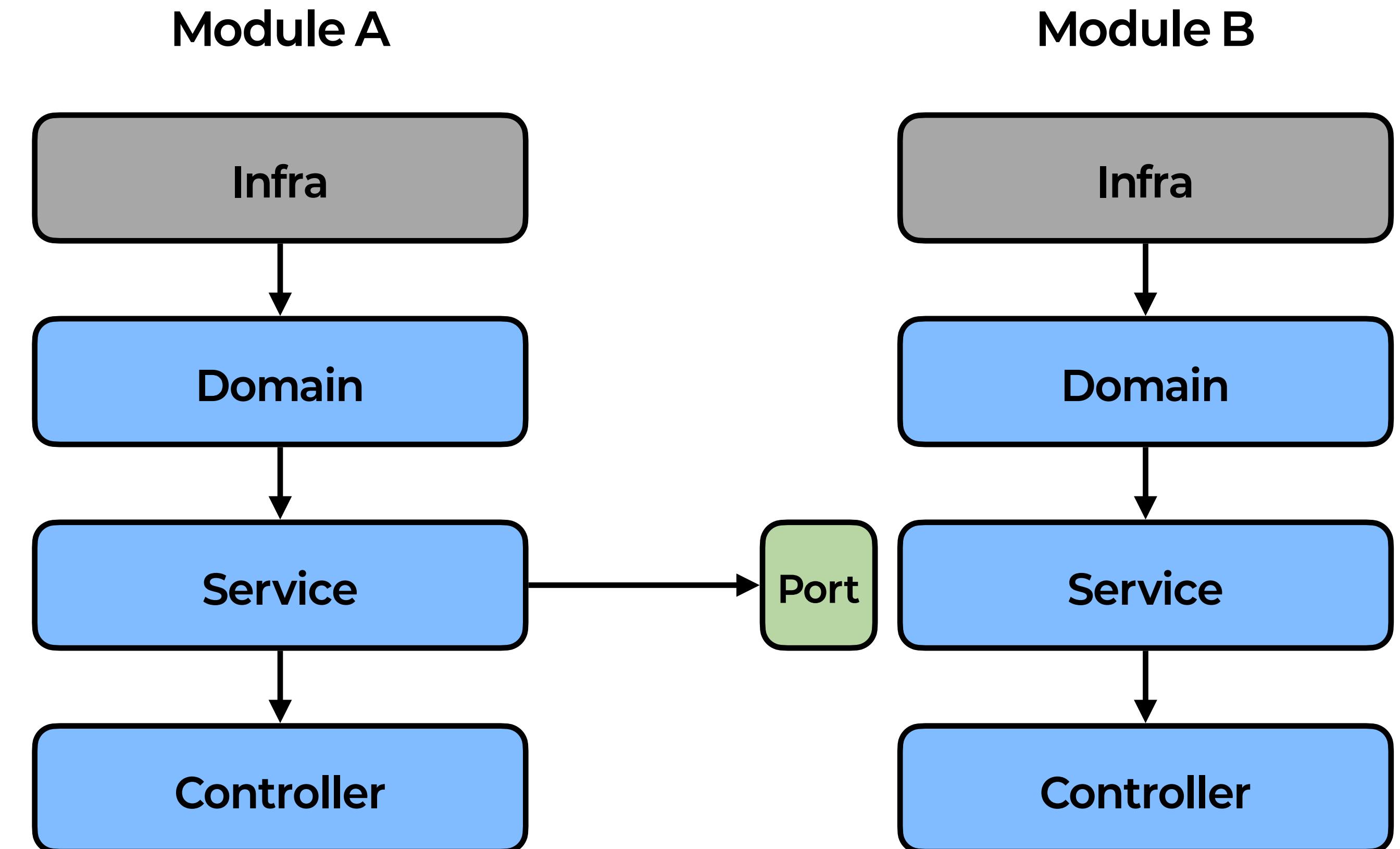
```
@Service
class UserService {
    fun registerUser(request: RegisterUserRequest): Effect<User, UserException> {
        return User.create(id = 1, username = request.username, email = request.email)
            .flatMap { createdUser →
                sendEmail(createdUser)
            .recover { createdUser }
        }
    }

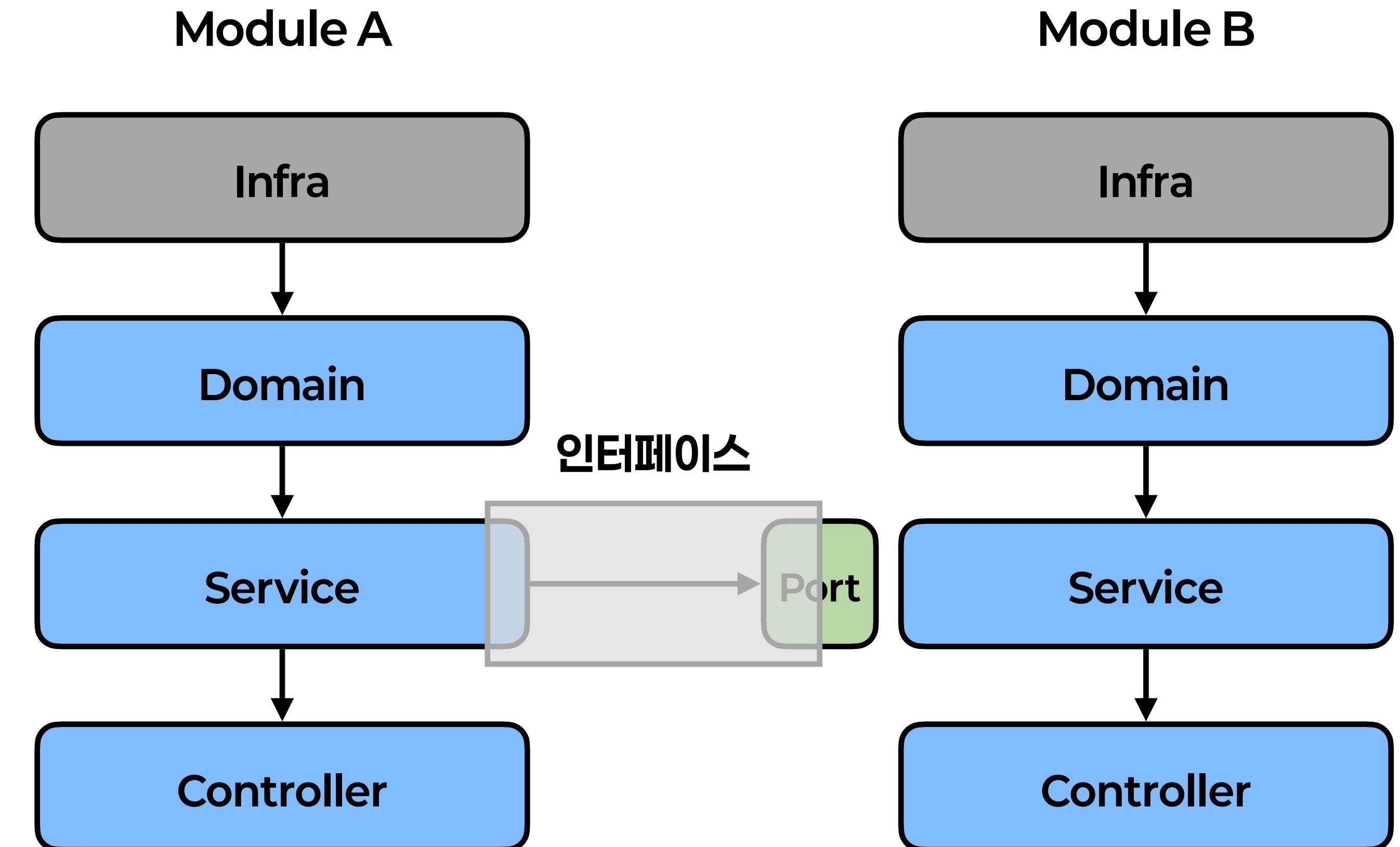
    private fun sendEmail(createdUser: User): Effect<User, EmailSendingException> = TODO()
}
```

서비스 레이어에선 **비즈니스 규칙**에 따라 로직을 작성할 수 있어요

```
@RestController
class UserController(private val userService: UserService) {
    fun registerUser(request: RegisterUserRequest): CommonResponse<RegisterUserResponse> {
        return userService.registerUser(request)
            .fold(
                onSuccess = { user ->
                    CommonResponse(
                        code = "200",
                        data = RegisterUserResponse(id = user.id, name = user.username.value),
                        message = "User registered successfully"
                    )
                },
                onFailure = { error ->
                    when (error) {
                        is ValidationException -> CommonResponse(
                            code = "400",
                            message = error.message ?: "Validation failed"
                        )
                        else -> throw error
                    }
                }
            )
    }
}
```

컨트롤러 레이어에서 결과에 따라 클라이언트에 보내줄 **응답**을 결정할 수 있어요



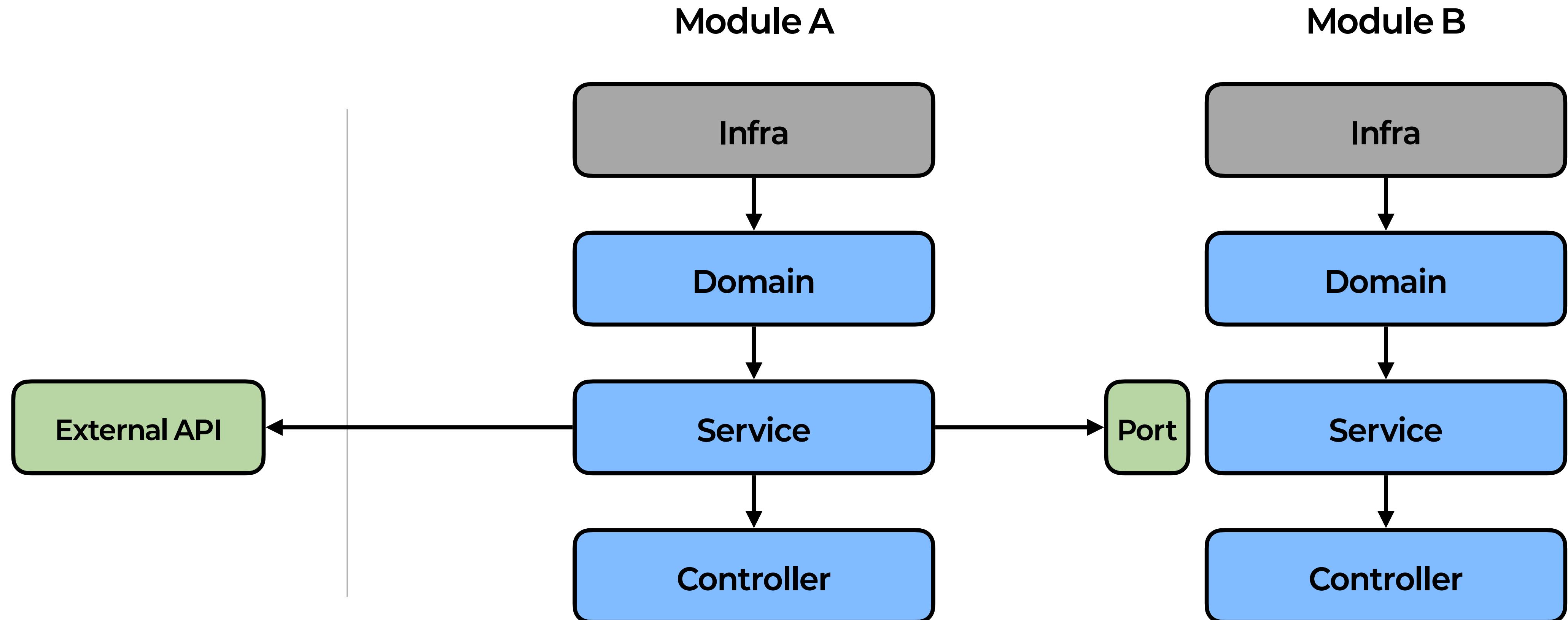


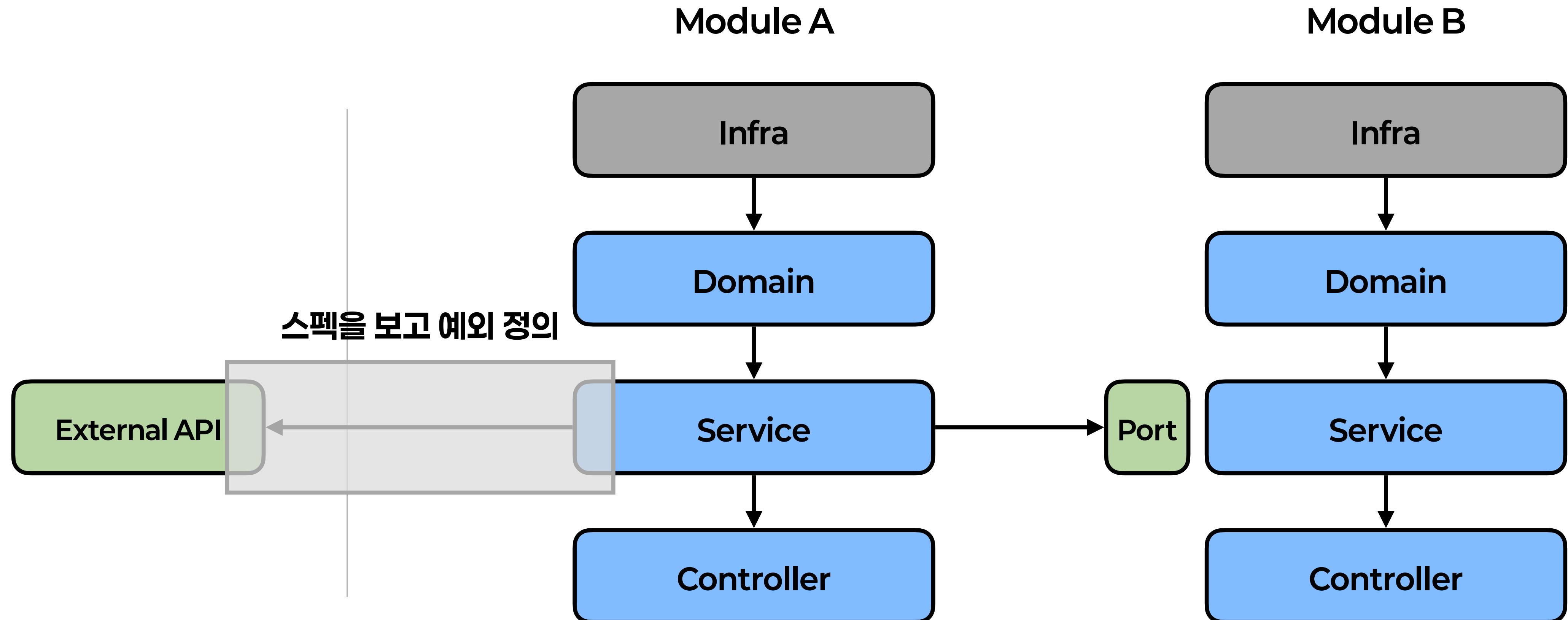
```
/**  
 * Email Module  
 */  
sealed class EmailException(override val message: String?): RuntimeException(message)  
data class EmailSendingException(override val message: String): EmailException(message)  
data class EmailApiTimeoutException(override val message: String): EmailException(message)  
  
interface EmailPort {  
    fun sendEmail(request: SendRequest): Effect<Email, EmailException>  
}  
  
@Service  
class EmailService: EmailPort {  
    override fun sendEmail(request: SendRequest): Effect<Email, EmailException> {  
        // ...  
    }  
}
```

다른 모듈의 인터페이스를 파악할 수 있어요

```
/**  
 * Email Module  
 */  
sealed class EmailException(override val message: String?): RuntimeException(message)  
data class EmailSendingException(override val message: String): EmailException(message)  
data class EmailApiTimeoutException(ov  
  
interface EmailPort {  
    fun sendEmail(request: SendRequest):  
}  
  
@Service  
class EmailService: EmailPort {  
    override fun sendEmail(request: Send  
        // ...  
    }  
}  
  
/**  
 * User Module  
 */  
@Service  
class UserService(private val emailPort: EmailPort) {  
    fun registerUser(request: RegisterUserRequest): Effect<User, RuntimeException> {  
        return User.create(id = 1, username = request.username, email = request.email)  
            .flatMap { createdUser →  
                emailPort.sendEmail(SendRequest(/* ... */))  
                    .recoverIf({ it is EmailSendingException}) { createdUser }  
                    .map { createdUser }  
            }  
    }  
}
```

다른 모듈의 인터페이스를 파악할 수 있어요





```
sealed class ExternalAPIException: RuntimeException() {  
    data class BadRequestException()  
    data class TimeoutException()  
    data class RateLimitException()  
    data class ServiceUnavailableException()  
    data class MaxLimitException()  
    // ...  
}
```

외부 API 스펙에 맞춰 가능한 예외를 정의하고 처리해야 해요

트랜잭션 처리

- @Transactional을 사용할 때는 주의해야 해요
- 모나드를 반환하면 **진짜 예외가 아닌 값**이기 때문에 Rollback이 발생하지 않아요
- 실패한 경우 다시 예외를 잡아서 던지거나 다음과 같은 확장 함수를 만들 수 있어요

```
fun <T, E: Throwable> Effect<T, E>.rollbackIfFailure(): Effect<T, E> {  
    if (this is Effect.Failure) {  
        try {  
            val transactionStatus = TransactionAspectSupport.currentTransactionStatus()  
            transactionStatus.setRollbackOnly()  
        } catch (e: NoTransactionException) {  
            // 트랜잭션이 없는 경우 무시  
        }  
    }  
    return this  
}
```

```
@Service
class UserService {
    @Transactional
    fun registerUser(req: RegisterUserRequest): Effect<User, Exception> {
        val result = validate(req)
            .flatMap { updateUser(it) }
            .flatMap { sendEmail(it) }

        return when (result) {
            is Effect.Success -> result.value
            is Effect.Failure -> {
                when (result.error) {
                    is EmailSendException -> result.error // 이메일 전송 실패는 롤백 안함
                    else -> throw result.error // Rollback
                }
            }
        }
    }
}
```

예외에 따른 트랜잭션 전략이 필요해요

@ControllerAdvice 처리

- 트랜잭션과 마찬가지로 모나드는 예외가 아니기 때문에 잡하지 않아요
- 실패한 경우 다시 예외로 던져야 할 필요가 있어요

마치며

Monad Comprehension

- 함수형적인 코드 작성이 익숙하지 않다면 로직 파이프를 구축하는 것이 어려울 수 있어요
- 특히 최적화된 코드를 작성하다 보면 중첩된 flatMap과 map이 생길 수 있어요
- 이를 함수형적으로 처리하면 가독성도 떨어지고 복잡할 수 있어요

```
val result = getUserId(1)
    .flatMap { user ->
        // 중첩!
        getAllPosts()
            .map { posts ->
                // user가 필요해서 nested하게 작성해야 해요
                posts.filter { it.userId == user.id }
            }
    }
```

Monad Comprehension

- Monad Comprehension을 이용하면 보편적인 방법으로 코드를 작성할 수 있어요
- 언어적으로 Kotlin은 Monad Comprehension을 지원하지 않지만
수신 객체(Context Receiver)와 확장 함수를 사용하여 흉내낼 수 있어요
- binding이라는 함수 내에서 bind() 함수를 통해 모나드를 벗겨낼 수 있어요
- 만약 실패하면 즉시 Effect가 반환돼요

```
val result: Effect<List<String>, Throwable> = binding {  
    val user = getUserId(1).bind()  
    val posts = getAllPosts().bind()  
    posts.filter { it.userId == user.id }.map { it.title }  
}
```

자바에서는?

- 이번 발표 자료에선 Kotlin으로만 설명했지만 Java에서도 ROP를 할 수 있어요
- Vavr라는 라이브러리를 사용하면 여러 모나드를 쉽게 사용할 수 있어요
 - Java에서 불변성과 함수형적인 예외 처리를 도와주는 라이브러리예요
 - Try, Either, Option과 같은 모나드 컨테이너를 제공해요
 - 2021년부터 업데이트가 중단되었지만, 2024년 10월부터 새로운 커미터가 활동을 재개했어요



Vavr의 Try<T> 사용 예제

- 앞서 살펴본 Effect처럼 성공과 실패로 나타낼 수 있는 모나드에요
- 내부적으로 try-catch를 사용하여 예외를 값으로 만들어요

```
public Integer riskyFunc(String input) throws NumberFormatException {  
    if ("fail".equals(input)) {  
        throw new RuntimeException("의도된 실패!");  
    }  
    return Integer.parseInt(input); // NumberFormatException 가능  
}
```

```
Try<Integer> result1 = Try.of(() -> riskyFunc("123")); // Success(123)  
Try<Integer> result2 = Try.of(() -> riskyFunc("abc")); // Failure(NumberFormatException)  
Try<Integer> result3 = Try.of(() -> riskyFunc("fail")); // Failure(RuntimeException)
```

ROP는 은총알이 아니에요

- 스택 트레이스나 예외가 발생한 위치가 중요한 경우
- 빠르게 실패해야 하는 경우
- 팀 내 합의가 없는 경우
- 성능이 매우 중요한 경우

예외 처리의 방향성

- 새롭게 탄생한 많은 언어가 try-catch 대신 값을 이용한 방법을 제공하고 있어요
 - Go, Rust, Zig ...
- 점점 예외 처리를 명시적으로 처리하는 방향으로 나아가고 있어요
- 그런 관점에서 레일웨이 지향 프로그래밍적인 방식을 알아두면 나쁘지 않을거에요



감사합니다

이선협 @kciter