# 6장. 단위 테스트 스타일



## 6.1 단위 테스트의 세 가지 스타일

- 출력 기반 테스트(output-based testing)
- 상태 기반 테스트(state-based testing)
- 통신 기반 테스트(communication-based testing)

## 6.1.1 출력 기반 테스트 정의

- 테스트 대상 시스템(SUT)에 입력을 넣고 `생성되는 출력`을 점검하는 방식
- 전역 상태나 내부 상태를 변경하지 않는 코드에만 적용되어 `반환 값만 검증` 하면 된다.

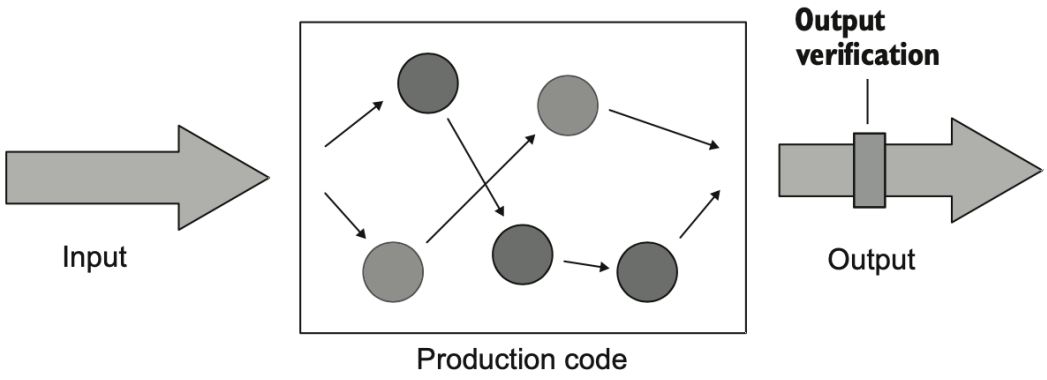


**Figure 6.1 In output-based testing, tests verify the output the system generates. This style of testing assumes there are no side effects and the only result of the SUT's work is the value it returns to the caller.**

### 예제

- calculateDiscount 메서드의 결과는 반환된 할인, 즉 출력 값 뿐이다.
- 출력 기반 단위 테스트 스타일은 함수형(functional)이라고도 한다.
- `사이드 이펙트 없는 코드` 선호를 강조하는 함수형 프로그래밍(functional programming) 에 뿌리를 둔다.

```
public class PriceEngine {
    public BigDecimal calculateDiscount(Product... products) {
        BigDecimal discount = new BigDecimal(products.length * 0.01);
        return discount.min(new BigDecimal("0.2"));
    }
}

@Test
public void discountOfTwoProducts() {
    Product product1 = new Product("Hand wash");
    Product product2 = new Product("Shampoo");
    PriceEngine sut = new PriceEngine();
    BigDecimal discount = sut.calculateDiscount(product1, product2);
    assertEquals(new BigDecimal("0.02"), discount);
}
```

## 6.1.2 상태 기반 스타일 정의

- 작업이 완료된 후 `시스템 상태`를 확인하는 방식
- `상태` 라는 용어는 SUT, 협력자 중 하나, 데이터베이스, 파일 시스템 등과 같은 프로세스 외부 의존성의 상태 등을 의미할 수 있다.

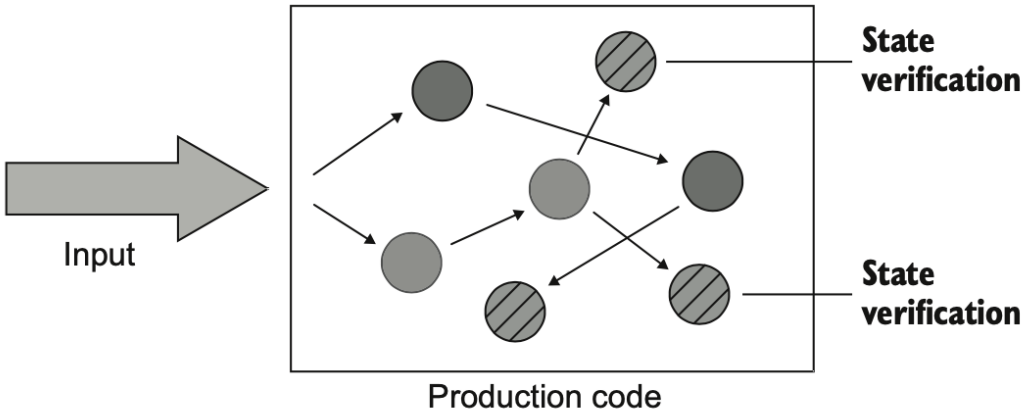


**Figure 6.3 In state-based testing, tests verify the final state of the system after an operation is complete. The dashed circles represent that final state.**

### 예제

- 상품을 추가한 후 Products 컬렉션을 검증한다.
- 출력 기반 테스트 예제와 달리 addProduct() 의 결과는 주문 상태의 변경이다.

```
public class Order {
    private List<Product> products = new ArrayList<>();

    public List<Product> getProducts() {
        return Collections.unmodifiableList(products);
    }

    public void addProduct(Product product) {
        products.add(product);
    }
}

@Test
public void addingAProductToAnOrder() {
    Product product = new Product("Hand wash");
    Order sut = new Order();
    sut.addProduct(product);
    assertEquals(1, sut.getProducts().size());
    assertEquals(product, sut.getProducts().get(0));
}
```

## 6.1.3 통신 기반 스타일 정의

- 목을 사용해 `테스트 대상 시스템과 협력자 간의 통신`을 검증하는 방식

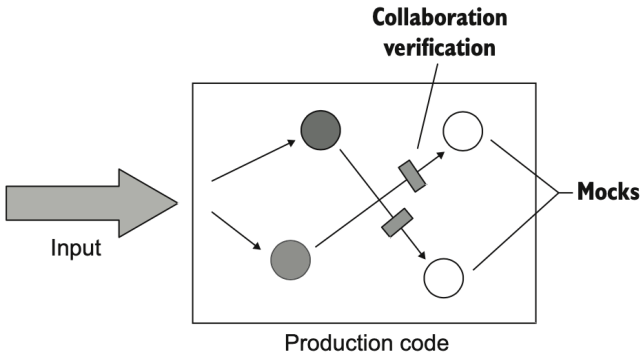


**Figure 6.4 In communication-based testing, tests substitute the SUT's collaborators with mocks and verify that the SUT calls those collaborators correctly.**

### 예제

- IEmailGateway.class 를 mocking 하여 객체 생성
- Controller 생성 후 greetUser 메서드를 호출하여 이메일을 전송
- IEmailGateway 의 sendGreetingsEmail 메서드가 1번 호출되었는지 검증

```
@Test
public void sendingAGreetingsEmail() {
    IEmailGateway emailGatewayMock = mock(IEmailGateway.class);
    Controller sut = new Controller(emailGatewayMock);
    sut.greetUser("user@email.com");
    verify(emailGatewayMock, times(1)).sendGreetingsEmail("user@email.com");
}
```

# 6.2 단위 테스트 스타일 비교

- 좋은 단위 테스트의 4대 요소로 비교한다.
  - 회귀 방지
  - 리팩토링 내성
  - 빠른 피드백
  - 유지 보수성

## 6.2.1 회귀 방지와 피드백 속도 지표로 스타일 비교하기

- 회귀 방지 지표는 3가지 특성으로 결정된다.
  1. 코드의 양
  2. 코드 복잡도
  3. 도메인 유의성

### 회귀 방지

- 다른 두 기반 테스트 스타일은 코드가 많든 적은 상관 없지만 `통신 기반 테스트에서 예외`가 있다.
  - 작은 코드 조각 검증, 모두 mocking 하는 등 오버 엔지니어링의 극단적 사례가 될 수 있다.

### 피드백 속도

- 테스트 스타일과 속도는 상관관계가 거의 없다.
- 테스트가 (프로세스 외부 의존성과 떨어져) 단위 테스트 영역에 있는 한, 모든 스타일은 테스트 실행 속도가 `거의 동일`하다.
- mock 은 런타임에 지연 시간이 생기는 편이므로 통신 기반 테스트가 약간 `나쁠` 수 있다.

## 6.2.2 리팩토링 내성 지표로 스타일 비교하기

### 리팩토링 내성이란?

- 리팩토링 중에 발생하는 `거짓 양성 수에 대한 척도` 이다.

### 출력 기반 테스트

- 테스트가 테스트 대상 메서드에서만 결합
  - 거짓 양성 `방지`가 가장 우수하다.
  - 구현 세부 사항에 결합하는 경우는 테스트 대상 메서드가 `구현 세부 사항일 때` 뿐이다.

### 상태 기반 테스트

- 거짓 양성이 되기 쉽다. `방지가 쉽지 않다.`
- 테스트 대상 메서드 외에도 클래스 상태와 함께 작동한다.
- `구현 세부 사항`에 테스트가 얽매일 가능성이 커진다.
  - API 노출 영역에 의존

### 통신 기반 테스트

- 허위 경보에 가장 취약하다. `방지가 어렵다.`
- 항상 스텁과 상호 작용하는 경우
- 상호 작용의 사이드 이펙트가 외부 환경에 보이는 경우에만 mock 이 괜찮다.
- **리팩토링 내성을 잘 지키려면** 통신 기반 테스트를 사용할 때 더 신중해야 한다.

### 통신 기반 테스트 - 거짓 양성 줄이는 방법

1. `캡슐화`
2. 테스트를 `식별있는 동작`에만 결합

## 6.2.3 유지 보수성 지표로 스타일 비교하기

- 하나 이상의 프로세스 외부 의존성과은 유지 보수가 어렵다.

### 특성

1. 테스트를 이해하기 얼마나 어려운가?(테스트 크기에 대한 함수)
2. 테스트를 실행하기 얼마나 어려운가?(외부 의존성 상태)

### 출력 기반 테스트의 유지 보수성

- 가장 유지 보수하기 `용이`하다.
- 코드가 짧고 간결
- `입력 공급`과 `출력 검증` 두 가지로 요약할 수 있다.
- 프로세스 외부 의존성을 다루지 않는다.

### 상태 기반 테스트의 유지 보수성

- 유지 보수하기 어렵다.
- 종종 출력 검증보다 더 많은 공간을 차지한다.

### 상태 기반 테스트의 유지 보수성 - 예제

- Article 객체를 통해 addComment 메서드를 호출하여 Comment 객체를 추가하고 올바르게 추가되었는지 검증한다.
- 추가된 Comment 의 필드 값들이 모두 예상한 값과 일치하는지 검증

```
@Test
public void addingACommentToAnArticle() {

    Article sut = new Article();
    String text = "Comment text";
    String author = "John Doe";
    LocalDateTime now = LocalDateTime.of(2019, 4, 1, 0, 0, 0);

    sut.addComment(text, author, now);

    assertEquals(1, sut.getComments().size());
    assertEquals(text, sut.getComments().get(0).getText());
    assertEquals(author, sut.getComments().get(0).getAuthor());
    assertEquals(now, sut.getComments().get(0).getDateCreated());
}
```

1. `헬퍼 메서드`로 문제 완화

```
@Test
public void addingACommentToAnArticle() {

    Article sut = new Article();
    String text = "Comment text";
    String author = "John Doe";
    LocalDateTime now = LocalDateTime.of(2019, 4, 1, 0, 0, 0);
    sut.addComment(text, author, now);

    // 헬퍼 메서드 사용한다.
    assertThat(sut).hasNumberOfComments(1)
                   .hasComment(text, author, now);
}
```

2. `동등 멤버 정의` 방식
    - 클래스가 값에 해당하고 값 객체로 변환할 수 있을 때만 효과적

```
@Test
public void addingACommentToAnArticle() {

    Article sut = new Article();
    // 동등 멤버 정의한다.
    Comment comment = new Comment("Comment text", "John Doe", LocalDateTime.of(2019, 4, 1, 0, 0));

    sut.AddComment(comment.getText(), comment.getAuthor(), comment.getDateCreated());

    // 동등 멤버로 비교한다.
    Assertions.assertThat(sut.getComments()).containsExactly(comment);
}
```

### 통신 기반 테스트의 유지 보수성

- 유지 보수성 지표에서 다른 테스트 보다 점수가 가장 낮다.
- 테스트 대역과 상호 작용 검증을 설정해야 해서 공간을 많이 차지한다.
- `mock-chain` 형태인 경우 유지 보수가 더 어려워진다.

## 6.2.4 스타일 비교하기 : 결론

| | 출력 기반 | 상태 기반 | 통신 기반 |
|---|---|---|---|
| 리팩터링 내성을 지키기 위한 노력 | 낮음 | 중간 | 중간 |
| 유지비 | 낮음 | 중간 | `높음` |

`→ 출력 기반 스타일를 선호하라.`

- 출력 기반 테스트는 함수형으로 작성된 코드에만 적용할 수 있다.

# 6.3 함수형 아키텍처 이해

## 6.3.1 함수형 프로그래밍이란?

- **함수의 정의**

```
[함수형 프로그래밍]은 수학적 함수를 사용한 프로그래밍이다.

[수학적 함수]는
1. 모든 입출력은 메서드 시그니처에 명시해야 한다.
2. 숨은 입출력이 없는 함수(또는 메서드)이다.
3. 호출 횟수에 상관없이 주어진 입력에 대해 동일한 출력을 생성한다.
```

### 수학적 함수

- 입출력을 명시한 수학적 함수는 테스트가 짧고 간결하며 이해하고 유지 보수하기 쉬워 테스트하기가 매우 쉽다.
- 출력 기반 테스트를 적용할 수 있는 메서드 유형은 수학적 함수뿐이다.
- 유지 보수성이 뛰어나고 거짓 양성 빈도가 낮다.

### 숨은 입출력

- 가독성이 떨어지고 테스트하기 힘들다.

### 숨은 입출력의 유형

- `사이드 이펙트`
  - 메서드 시그니처에 표시되지 않은 출력
  - 연산은 클래스 인스턴스의 상태를 변경하고 디스크의 파일을 업데이트하는 등 사이드 이펙트를 발생시킨다.
- `예외`
  - 호출된 예외가 호출 스택의 어느 곳에서도 발생할 수 있다.
  - 메서드 시그니처가 전달하지 않는 출력을 추가하게 된다.
- `내외부 상태에 대한 참조`
  - DateTime.Now 와 같이 정적 속성을 사용해 현재 날짜와 시간을 가져오는 메서드가 있는데 이는 데이터베이스에서 데이터를 질의할 수 있고 비공개 변경 가능 필드(private 필드)를 참조할 수도 있다.
  - 없는 실행 흐름에 대한 입력이며 따라서 숨은 입출력이라고 판단한다.

### 메서드가 수학적 함수인지 판별하는 가장 좋은 방법

- 프로그램 동작을 변경하지 않고 해당 메서드에 대한 호출을 반환 값으로 대체할 수 있는지 확인한다.
- 수학적 함수의 예시

```
public int Increment(int x)
{
  return x + 1;
}

int y = Increment(4);
int y = 5;
```

- 수학적 함수가 아닌 예시

```
int x = 0;
public int Increment()
{
  x++;    // 숨은 출력, x의 변경이 발생한다.
  return x;
}
```

```
public Comment AddComment(string text)
{
```

```
    var comment = new Comment(text);
    _comments.Add(comment);   // 사이드 이펙트(추가적인 숨은 출력)가 발생하는 부분
     return comment;
}
```

## 6.3.2 함수형 아키텍처란?

### 함수형 프로그래밍의 목표

- `비즈니스 로직을 처리하는 코드와 사이드 이펙트를 일으키는 코드를 분리`하는 것이다.

### 비즈니스 로직과 사이드 이펙트를 분리하는 두 가지 코드 유형

1. 결정을 내리는 코드
   - 수학적 함수를 사용해서 작성한다.
   - `함수형 코어` 또는 `불변 코어`라고도 한다.
2. 해당 결정에 따라 작용하는 코드
   - 데이터베이스의 변경이나 메시지 버스로 전송된 메시지와 같이 가시적인 부분으로 변환
   - `가변 셸`이라고도 한다.

### 함수형 코어와 가변 셸

- `가변 셸`을 모든 입력을 수집한다.
- `함수형 코어`는 결정을 생성한다.
- `셸`은 결정을 사이드 이펙트로 변환한다.
- 결정을 나타내는 클래스(함수형 코어)에 정보가 충분히 있는지 확인해야 한다.
- 가변 셸은 아무말도 하지 않아야 한다.
- **여기서의 목표는 출력 기반 테스트로 함수형 코어를 다루고, 가변 셸을 훨씬 더 적은 수의 통합 테스트에 맡기는 것이다.**

## 6.3.3 함수형 아키텍처와 육각형 아키텍처 비교

### 함수형 아키텍처와 육각형 아키텍처의 공통점

- `관심사 분리` 아이디어를 기반으로 한다.
- 의존성 간 단방향 흐름이다.
- 도메인 계층의 클래스는 서로에게 의지하되, 애플리케이션 서비스 계층의 클래스에 의존하지 않는다.
- 함수형 코어(불변 코어)는 가변 셸에 의존하지 않는다.
- Hexagonal
  - 도메인 계층은 `비즈니스 로직`, 애플리케이션 서비스 계층은 `외부 애플리케이션과의 통신`에 책임이 있다.
- Functional

- `결정`과 `실행`의 책임을 분리한다.

### 함수형 아키텍처와 육각형 아키텍처의 차이점

- 사이드 이펙트 처리 방식이 다르다.
  - 함수형 아키텍처는
    - 불변 코어에서 가변 셸에 의존하지 않는다.
    - 가변 셸에서 불변 코어를 분리시켜 셸이 제공하는 입력을 단순한 값으로 모방할 수 있다.
  - 육각형 아키텍처는
    - 모드는 수정 사항은 도메인 계층 내에 있어야 한다.
    - 계층의 경계를 넘어서는 안된다.
    - 도메인 클래스 인스턴스는 데이터베이스에 직접 저장할 수 없지만 상태는 변경 가능하고 애플리케이션 서비스에서 변경 사항을 데이터베이스에 반영한다.

# 6.4 함수형 아키텍처와 출력 기반 테스트로의 전환

### 함수형 아키텍처로의 리팩터링

1. 프로세스 외부 의존성에서 목으로 변경
2. 목에서 함수형 아키텍처로 변경

(외부 의존성 → mock → 함수형 아키텍처)

## 6.4.1 감사 시스템 소개

- 텍스트 기반 파일을 기반 저장소로 사용
- 가장 최근 파일의 마지막 줄에 방문자 이름과 방문 시간을 추가한다.
- 파일당 최대 항목 수에 도달하면 인덱스를 증가시켜 새 파일을 작성한다.

```
public class AuditManager {
    private final int _maxEntriesPerFile;
    private final String _directoryName;

    public AuditManager(int maxEntriesPerFile, String directoryName) {
        _maxEntriesPerFile = maxEntriesPerFile;
        _directoryName = directoryName;
    }

    public void addRecord(String visitorName, Date timeOfVisit) throws IOException {
        String[] filePaths = new File(_directoryName).list();
        (int index, String path)[] sorted = sortByIndex(filePaths);
        String newRecord = visitorName + ";" + timeOfVisit;
        if (sorted.length == 0) {
            String newFile = Paths.get(_directoryName, "audit_1.txt").toString();
            Files.write(Paths.get(newFile), newRecord.getBytes());
            return;
        }
        (int currentFileIndex, String currentFilePath) = sorted[sorted.length - 1];
        List<String> lines = Files.readAllLines(Paths.get(currentFilePath));
        if (lines.size() < _maxEntriesPerFile) {
            lines.add(newRecord);
            String newContent = String.join(System.lineSeparator(), lines);
            Files.write(Paths.get(currentFilePath), newContent.getBytes());
        } else {
            int newIndex = currentFileIndex + 1;
```

```
            String newName = "audit_" + newIndex + ".txt";
            String newFile = Paths.get(_directoryName, newName).toString();
            Files.write(Paths.get(newFile), newRecord.getBytes());
        }
    }

    private (int, String)[] sortByIndex(String[] filePaths) {
        List<Integer> indices = new ArrayList<>();
        List<String> paths = new ArrayList<>();
        for (String path : filePaths) {
            String filename = new File(path).getName();
            if (filename.matches("^audit_\\d+\\.txt$")) {
                int index = Integer.parseInt(filename.replaceAll("^audit_(\\d+)\\.txt$", "$1"));
                indices.add(index);
                paths.add(path);
            }
        }
        int[] sortedIndices = indices.stream().mapToInt(i -> i).sorted().toArray();
        (int, String)[] sorted = new (int, String)[sortedIndices.length];
        for (int i = 0; i < sortedIndices.length; i++) {
            int index = sortedIndices[i];
            String path = paths.get(indices.indexOf(index));
            sorted[i] = new Tuple<>(index, path);
        }
        return sorted;
    }
}
```

- 작업 디렉토리에서 전체 파일 목록을 검색한다.
- 인덱스별 정렬한다.
- 감사 파일이 없으면 단일 레코드로 첫 번째 파일을 생성한다.
- 감사 파일이 있으면 최신 파일을 가져와서 파일 항목 수 한계에 도달했는지 확인하고 새 레코드 또는 새 파일을 추가한다.

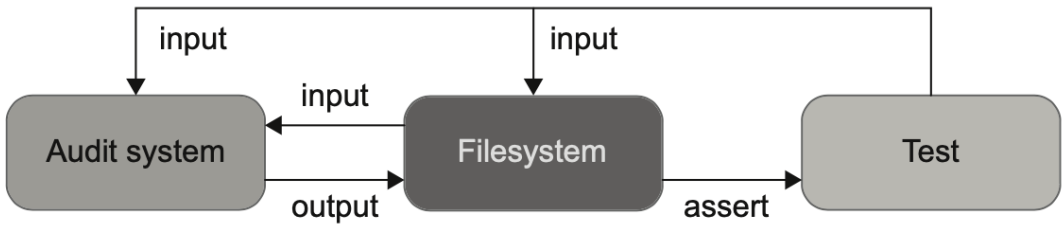


**Figure 6.12 Tests covering the initial version of the audit system would have to work directly with the filesystem.**

- (감사 시스템 초기 버전에 대한 테스트는 파일 시스템에 직접적으로 수행해야 한다.)

- 좋은 단위 테스트 4대 요소에 대한 점수

| | 초기 버전 |
|---|---|
| 회귀 방지 | 좋음 |
| 리팩토링 내성 | 좋음 |
| 빠른 피드백 | 나쁨 |
| 유지 보수성 | 나쁨 |

- 파일 시스템에 직접 작동하는 테스트는 단위 테스트의 정의에 맞지 않는다.
- 단위 테스트는
  - 단일 동작 단위를 검증한다.
  - 빠르게 수행한다.
  - 다른 테스트와 별도로 처리한다.

## 6.4.2 테스트를 파일 시스템에서 분리하기 위한 목 사용

- 일반적으로 파일 시스템을 목으로 처리해서 해결한다.
- 모든 연산을 별도의 클래스(IFileSystem)로 도출하고 AuditManager에 생성자로 해당 클래스를 주입한다.

```
public interface FileSystem {
    String[] GetFiles(String path);
    void WriteAllText(String path, String contents);
    List<String> ReadAllLines(String path);
}

public class AuditManager {
    private final int maxEntriesPerFile;
    private final String directoryName;
    private final FileSystem fileSystem;

    public AuditManager(int maxEntriesPerFile, String directoryName, FileSystem fileSystem) {
        this.maxEntriesPerFile = maxEntriesPerFile;
        this.directoryName = directoryName;
        this.fileSystem = fileSystem;
    }
}
```

  - addRecord 메서드

```
public class AuditManager {
    private final int maxEntriesPerFile;
    private final String directoryName;
    private final IFileSystem fileSystem;

    public AuditManager(int maxEntriesPerFile, String directoryName, IFileSystem fileSystem) {
        this.maxEntriesPerFile = maxEntriesPerFile;
        this.directoryName = directoryName;
        this.fileSystem = fileSystem;
    }

    public void addRecord(String visitorName, LocalDateTime timeOfVisit) {
        String[] filePaths = fileSystem.getFiles(directoryName);
        List<Path> sorted = sortByIndex(filePaths);
        String newRecord = visitorName + ';' + timeOfVisit.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
        if (sorted.isEmpty()) {
            String newFile = Paths.get(directoryName, "audit_1.txt").toString();
            fileSystem.writeAllText(newFile, newRecord);
            return;
        }
        Path currentFilePath = sorted.get(sorted.size() - 1);
        List<String> lines = fileSystem.readAllLines(currentFilePath.toString());
        if (lines.size() < maxEntriesPerFile) {
            lines.add(newRecord);
            String newContent = String.join("\r\n", lines);
            fileSystem.writeAllText(currentFilePath.toString(), newContent);
        } else {
            int newIndex = Integer.parseInt(currentFilePath.getFileName().toString().replaceAll("\\D+", "")) + 1;
            String newName = "audit_" + newIndex + ".txt";
```

```
            String newFile = Paths.get(directoryName, newName).toString();
            fileSystem.writeAllText(newFile, newRecord);
        }
    }

    private List<Path> sortByIndex(String[] filePaths) {
        List<Path> paths = new ArrayList<>();
        for (String filePath : filePaths) {
            Path path = Paths.get(filePath);
            if (path.getFileName().toString().matches("audit_\\d+.txt")) {
                paths.add(path);
            }
        }
        return paths.stream()
                .sorted(Comparator.comparing(p -> Integer.parseInt(p.getFileName().toString().replaceAll("\\D+", ""))))
                .collect(Collectors.toList());
    }
}
```

- 공유 의존성을 없애고 테스트를 독립적으로 실행할 수 있는 테스트
  - 현재 파일의 항목 수가 한계에 도달했을 때 새 파일을 생성하는지 검증한다.
  - 애플리케이션은 최종 사용자가 볼 수 있는 파일을 생성한다.

```
@Test
public void A_new_file_is_created_when_the_current_file_overflows() {
    IFileSystem fileSystemMock = mock(IFileSystem.class);
    when(fileSystemMock.GetFiles("audits")).thenReturn(new String[]{
            "audits/audit_1.txt",
            "audits/audit_2.txt"
    });
    when(fileSystemMock.ReadAllLines("audits/audit_2.txt")).thenReturn(Arrays.asList(
            "Peter;2019-04-06T16:30:00",
            "Jane;2019-04-06T16:40:00",
            "Jack;2019-04-06T17:00:00"
    ));
    AuditManager sut = new AuditManager(3, "audits", fileSystemMock);
    sut.AddRecord("Alice", LocalDateTime.parse("2019-04-06T18:00:00"));
    verify(fileSystemMock).WriteAllText(
            "audits/audit_3.txt",
            "Alice;2019-04-06T18:00:00");
}
```

- 감사 시스템 초기 버전과 목을 사용한 버전 비교

| | 초기 버전 | 목 사용 |
|---|---|---|
| **회귀 방지** | 좋음 | 좋음 |
| **리팩토링 내성** | 좋음 | 좋음 |
| **빠른 피드백** | 나쁨 | 좋음 |
| **유지 보수성** | 나쁨 | 중간 |

## 6.4.3 함수형 아키텍처로 리팩터링 하기

- 인터페이스 뒤로 사이드 이펙트를 숨긴다.
- 해당 인터페이스를 AuditManager 에 주입한다.
- 대신, 사이드 이펙트를 클래스 외부로 완전히 이동시킨다.
- AuditManager 는 파일에 수행할 작업을 둘러싼 결정만 책임지면 된다.

- 새로운 클래스인 `Persister` 는 그 결정에 따라 파일 시스템에 업데이트를 적용하면 된다.

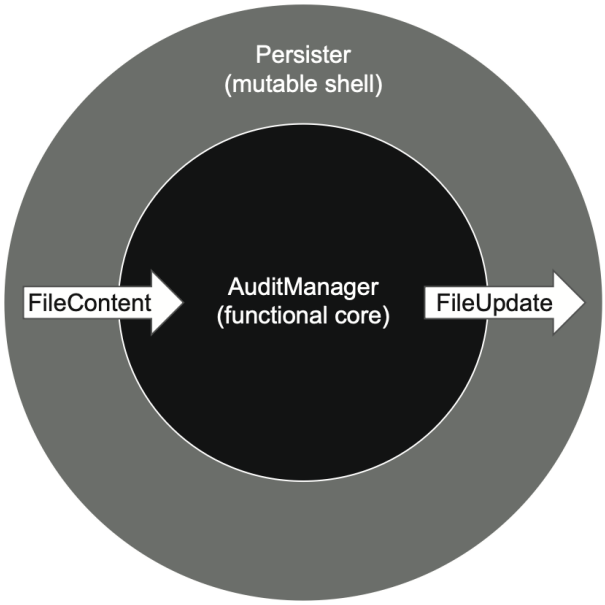


**Figure 6.14 `Persister` and `AuditManager` form the functional architecture. `Persister` gathers files and their contents from the working directory, feeds them to `AuditManager`, and then converts the return value into changes in the filesystem.**

- Persister 는 가변 셸 역할을 한다.
- AuditManager 는 함수형 코어 에 해당한다.

- 리팩터링 후의 AuditManager 예시
  - AuditManager 는 작업 디렉토리 경로 대신 FileContent 배열을 받는다.

```
public class AuditManager {
    private final int maxEntriesPerFile;

    public AuditManager(int maxEntriesPerFile) {
        this.maxEntriesPerFile = maxEntriesPerFile;
    }

    public FileUpdate addRecord(List<FileContent> files, String visitorName, DateTime timeOfVisit) {
        FileContent[] sorted = files.stream()
                .sorted(Comparator.comparingInt(fc -> Integer.parseInt(fc.getFileName().replaceAll("[^0-9]", ""))))
                .toArray(FileContent[]::new);
        String newRecord = visitorName + ";" + timeOfVisit;
        if (sorted.length == 0) {
            return new FileUpdate("audit_1.txt", newRecord);
        }
        FileContent currentFile = sorted[sorted.length - 1];
        List<String> lines = currentFile.getLines();
        if (lines.size() < maxEntriesPerFile) {
            lines.add(newRecord);
            String newContent = lines.stream().collect(Collectors.joining("\r\n"));
            return new FileUpdate(currentFile.getFileName(), newContent);
        } else {
            int newIndex = Integer.parseInt(currentFile.getFileName().replaceAll("[^0-9]", "")) + 1;
            String newName = "audit_" + newIndex + ".txt";
            return new FileUpdate(newName, newRecord);
        }
    }
}
```

- 리팩터링 후 FileContent
  - 결정을 내리기 위해 파일 시스템에 대해 알아야 할 모든 것을 포함한다.

```
public class FileContent {
    public final String fileName;
    public final String[] lines;

    public FileContent(String fileName, String[] lines) {
        this.fileName = fileName;
        this.lines = lines;
    }
}
```

- 리팩터링 후 FileUpdate
  - AuditManager 는 수행하려는 사이드 이펙트에 대한 명령을 반환한다. (상단 예시 참고)

```
public class FileUpdate {
    public final String fileName;
    public final String newContent;

    public FileUpdate(String fileName, String newContent) {
        this.fileName = fileName;
        this.newContent = newContent;
    }
}
```

- 리팩터링 후 Persister
  - 작업 디렉터리에서 내용을 읽어온 후 AuditManager 에게서 받은 업데이트 명령을 작업 디렉터리에 다시 수행하기만 하면 된다.

```
public class Persister {
    public FileContent[] readDirectory(String directoryName) throws IOException {
        return Files.list(Path.of(directoryName))
                .filter(Files::isRegularFile)
                .map(path -> {
                    try {
                        String fileName = path.getFileName().toString();
                        String[] lines = Files.readAllLines(path).toArray(new String[0]);
                        return new FileContent(fileName, lines);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                })
                .toArray(FileContent[]::new);
    }

    public void applyUpdate(String directoryName, FileUpdate update) throws IOException {
        String filePath = Path.of(directoryName, update.getFileName()).toString();
        Files.write(Path.of(filePath), update.getNewContent().getBytes());
    }
}
```

- 위의 예제들을 보면 모든 복잡도는 AuditManager 클래스에 있다. 이것이 `비즈니스 로직과 사이드 이펙트의 분리`이다.

### 육각형 아키텍처 적용해보기

- AuditManager 와 Persister 를 붙이기 위해 `애플리케이션 서비스` 클래스가 필요하다.

- 애플리케이션 서비스는 외부 클라이언트를 위한 시스템의 진입점을 제공한다.
- 감사 시스템의 동작을 쉽게 확인할 수 있다.
- **ApplicationService.java 포함 예시**

```
public class ApplicationService {
    private final String directoryName;
    private final AuditManager auditManager;
    private final Persister persister;

    public ApplicationService(String directoryName, int maxEntriesPerFile) {
        this.directoryName = directoryName;
        this.auditManager = new AuditManager(maxEntriesPerFile);
        this.persister = new Persister();
    }

    public void AddRecord(String visitorName, DateTime timeOfVisit) throws IOException {
        FileContent[] files = persister.ReadDirectory(directoryName);
        FileUpdate update = auditManager.AddRecord(files, visitorName, timeOfVisit);
        persister.ApplyUpdate(directoryName, update);
    }
}

public class AuditManager {
    private final int maxEntriesPerFile;

    public AuditManager(int maxEntriesPerFile) {
        this.maxEntriesPerFile = maxEntriesPerFile;
    }

    public FileUpdate AddRecord(FileContent[] files, String visitorName, DateTime timeOfVisit) {
        List<FileContent> sorted = Arrays.stream(files)
                .sorted(Comparator.comparingInt(f -> Integer.parseInt(f.FileName.split("_")[1].split("\\.")[0])))
                .collect(Collectors.toList());
        String newRecord = visitorName + ";" + timeOfVisit.toString();
        if (sorted.size() == 0) {
            return new FileUpdate("audit_1.txt", newRecord);
        }

        FileContent currentFile = sorted.get(sorted.size() - 1);
        List<String> lines = new ArrayList<>(Arrays.asList(currentFile.Lines));

        if (lines.size() < maxEntriesPerFile) {
            lines.add(newRecord);
            String newContent = String.join(System.lineSeparator(), lines);
            return new FileUpdate(currentFile.FileName, newContent);
        } else {
            int newIndex = Integer.parseInt(currentFile.FileName.split("_")[1].split("\\.")[0]) + 1;
            String newName = "audit_" + newIndex + ".txt";
            return new FileUpdate(newName, newRecord);
        }
    }
}

public class Persister {
    public FileContent[] ReadDirectory(String directoryName) throws IOException {
        return Files.walk(Paths.get(directoryName))
                .filter(Files::isRegularFile)
                .map(file -> {
                    try {
                        return new FileContent(file.getFileName().toString(), Files.readAllLines(file));
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                })
                .toArray(FileContent[]::new);
    }

    public void ApplyUpdate(String directoryName, FileUpdate update) throws IOException {
        String filePath = Paths.get(directoryName, update.FileName).toString();
        Files.write(Paths.get(filePath), update.NewContent.getBytes());
    }
}

public class FileContent {
    public final String FileName;
    public final String[] Lines;
```

```
    public FileContent(String fileName, String[] lines) {
        FileName = fileName;
        Lines = lines;
    }
}

public class FileUpdate {
    public final String FileName;
    public final String NewContent;

    public FileUpdate(String fileName, String newContent) {
        FileName = fileName;
        NewContent = newContent;
    }
}
```

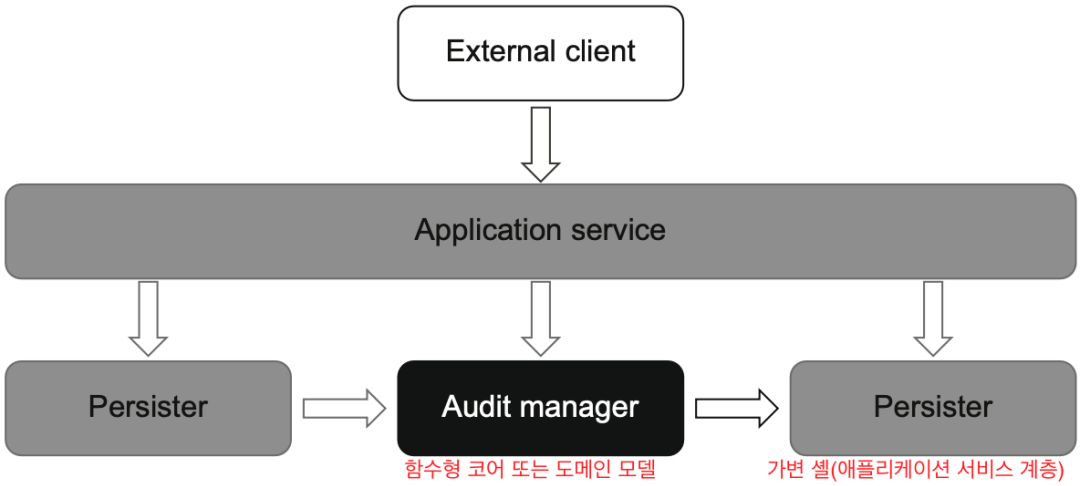


**Figure 6.15 `ApplicationService` glues the functional core (`AuditManager`) and the mutable shell (`Persister`) together and provides an entry point for external clients. In the hexagonal architecture taxonomy, `ApplicationService` and `Persister` are part of the application services layer, while `AuditManager` belongs to the domain model.**

- 이전 두 버전과 출력 기반 테스트를 비교

| | 초기 버전 | 목 사용 | 출력 기반 |
|---|---|---|---|
| 회귀 방지 | 좋음 | 좋음 | 좋음 |
| 리팩토링 내성 | 좋음 | 좋음 | 좋음 |
| 빠른 피드백 | 나쁨 | 좋음 | 좋음 |
| 유지 보수성 | 나쁨 | 중간 | 좋음 |

## 6.4.4 예상되는 추가 개발

- p.219 참고

# 6.5 함수형 아키텍처의 단점 이해하기

## 6.5.1 함수형 아키텍처 적용 가능성

- 실행 흐름이 간단하지 않음
- 의사 결정 절차의 중간 결과에 따라 프로세스 외부 의존 성에서 추가 데이터를 질의할 수 있다.
- 숨은 입력이 생길 수 있다.

- 데이터 베이스로 인해 숨은 입력이 생겼다.
- 더이상 함수형 아키텍처를 따르지 않음

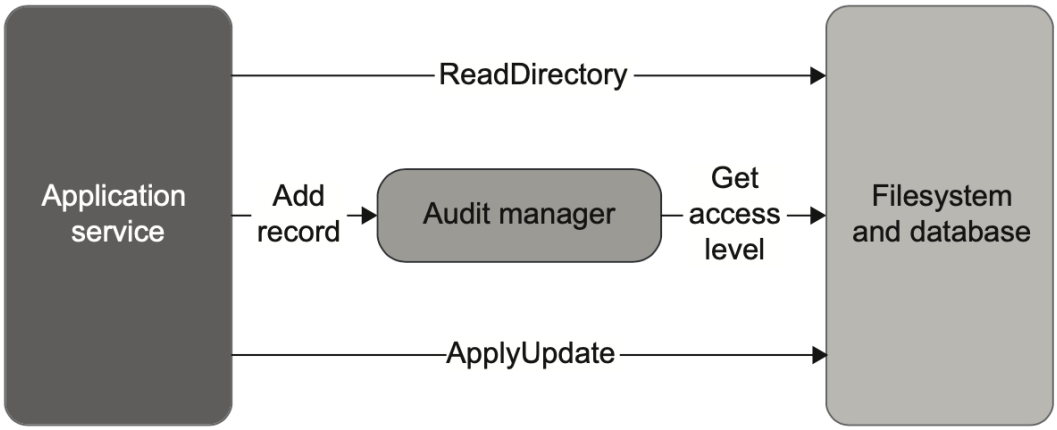


**Figure 6.16 A dependency on the database introduces a hidden input to `AuditManager`. Such a class is no longer purely functional, and the whole application no longer follows the functional architecture.**

### 두 가지 해결책

- 애플리케이션 서비스 전면에서 디렉터리 내용과 더불어 방문자 접근 레벨을 수집할 수 있다.
- AuditManager 에서는 IsAccessLevelCheckRequired() 와 같은 새로운 메서드를 둬야한다.

### 두 가지 해결책의 단점

- 첫번째 방법은 성능 저하가 우려된다.
  - 접근 레벨이 필요없어도 데이터베이스에 질의한다.
  - 비즈니스 로직과 외부 시스템과의 통신을 완전히 분리할 수 있다.
- 두번째 방법은 분리를 다소 완화한다.
  - 데이터베이스 호출에 대한 결정권이 애플리케이션 서비스로 넘어간다.

- 도메인 모델(AuditManager)을 데이터베이스에 의존시키는 건 좋은 방법이 아니다.

## 6.5.2 성능 단점

- 성능은 함수형 아키텍처의 흔한 논쟁이다.
- 성능과 코드 유지 보수성 간의 절충이다.

### 성능 이슈 정리

- 성능 영향이 눈에 띄지 않는다면? `함수형 아키텍처 + 유지 보수성 향상`
- 성능이 중요하다면? `전통적 아키텍처`를 따른다.

## 6.5.3 코드 베이스 크기 증가

- 함수형 아키텍처는 결정과 실행을 분리하면서 궁극적으로 코드 복잡도가 낮아지고 유지 보수성이 향상되지만 초기에 코딩이 더 필요하다는 것!
- 어떤 코드베이스는 너무 단순하고 비즈니스 관점에서 중요하지 않을 수 있다.
  - 이런 경우 함수형 아키텍처가 별 의미 없을 수 있다.
  - 전략적으로 적용하자.
- 비용이 많이 든다면 함수형 아키텍처를 따르지 말라.
  - 모든 도메인을 불변으로 할 수 없다.
  - 대부분 출력 기반과 상태 기반 스타일을 조합한다. (여기에 통신 기반 스타일 약간)
- 가능한 한 많은 테스트를 전환하라는 것이지 모든 테스트를 출력 기반 스타일로 전환하라는 것이 아니다.