

졸업프로젝트

TraceRecoviz

현재까지의 진행사항

1. 개요

추적성이 유지되던 프로젝트 상황



Unit code
version 1.xx



Test code
version 1.xx

추적성이 끊긴 프로젝트 상황



Unit code
version ??



Test code
version ??

2. 추적성 탐지를 위해 표현되어야 할 필수요소

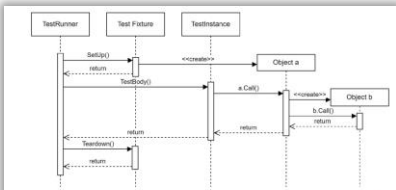


그림 1. xUnit 기반 Test Framework의 테스트 수행 절차

표 1. 시각화에 필요한 GoogleTest 개념

테스트 구성요소	역할
Fixture	테스트에 필요한 데이터나 환경을 준비
Stub	특정 기능을 dummy 객체로 대체하는 부분
Assertion	일일값과 기댓값을 명시
Parameterized Test	매개변수를 사용하여 같은 테스트를 여러 데이터로 실행하는 것을 식별
Test Case, Test Suite	특정한 기능을 검증하는 여러 개의 테스트가 어떤 그룹에 속하는지 표시
EventListener 등	이 외에 테스트 프레임워크에서 사용되는 객체들

그림 1. Test Framework 절차 및 테스트 구성요소

표 2. 도구별 시각화 구성요소 지원현황

시각화 구성요소	SequenceDiagram for C++ (JetBrains Plugin)	UModel (Altova)	clang-uml	DYNO	Enterprise Architect
분석 방식	정적	정적	정적	동적	동적
메서드 호출 흐름	○	X	○	○	○
조건문 및 분기 구조 표시	○	○	○	X	X
오브젝트	X	X	○	X	X
객체 라이프 사이클 표시	X	X	X	○	○
객체별 라이프라인 표시	X	X	X	X	X
GoogleTest 테스트 구성요소 표시	X	X	X	X	X
인자값, 반환값 표시	X	X	X	X	X

그림 2. 도구별 시각화 구성요소 지원현황

3. 새로운 도구의 필요성

- 테스트 코드의 시나리오 기반 시각화
- 기존도구는 표현하지 못하는 시각화 구성요소 포함

단위 테스트-코드 간 추적성 복구를 위한 시나리오 기반 시각화 연구

A Study on Scenario-Based Visualization Between Unit Test

요약
테스트 시스템에서는 개발 산출물이 최신 상태로 유지되지 않거나 이미 존재하지 않는 경우, 코드와 수신했던 변경사항에 단위 테스트에 반영되지 않는 문제가 발생한다. 단위 테스트와 단위 코드 간 추적성의 단절은 결과적으로 테스트가 어떤 의도한 기능을 검증하는지를 판단하기 어렵게 만든다. 단위 테스트가 통과되었고도 실제 해당 기능이 의도대로 동작하지 않는 상황을 방치할 수 있다. 개발자는 테스트 결과에 대해 신뢰하기 어렵다 [1]. 이를 해결하기 위해서는 개발자가 직접 코드와 테스트 간의 비교 및 분석을 해야 하며, 이는 많은 시간과 노력을 필요로 한다.

본 연구에서는 단위 테스트의 구조와 동작 흐름에 추적해야 할 필수 기능을 탐지하여 해당 흐름을 시각화하는 것을 목표로 한다. 이를 위해 UML(Unified Modeling Language) 표준 다이어그램으로, 객체 간의 상호작용을 시간 순서에 따라 표현하여 테스트의 실행 흐름을 분석하는 데 효과적으로 활용할 수 있다.

본 논문은 총 4장으로 구성된다. 2장에서는 본 연구의 적용 대상인 C++의 언어적 특징과 Google Test의 구조를 설명하고, 시각화에 필요한 구성 요소를 정리한다. 3장에서는 관련된 정적 분석 도구들의 동적 분석 도구들의 기능을 조사하고, 추적성 복구 측면에서의 활용 여부를 분석한다. 4장에서는 기존의 정적 분석 도구들과 비교하여, 시나리오 기반 추적성 복구를 위한 향후 연구 방향을 제시한다.

2. 단위 테스트 시각화를 위한 분석 대상

개발 언어 C++ 및 Google Test 환경을 대상으로 시각화에 필요한 요소를 살펴본다. C++는 정적 타이핑과 정적 타입 체킹을 가지며, 객체 지향 프로그래밍 패러다임을 채택하고 있다. 테스트 코드는 단위 코드와 함께 작성되며, 단위 코드의 실행을 검증하는 역할을 한다. Google Test는 C++의 테스트 프레임워크로, 단위 테스트를 위한 다양한 기능을 제공한다. 그러나 이 도구는 C++의 언어적 특징을 충분히 반영하지 못하며, 단위 코드의 실행 흐름을 시각화하는 데 한계가 있다. 따라서, 단위 테스트와 단위 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

3. 기존 시각화 도구 분석

시각화 도구는 정적 분석 도구(static analysis tool)와 동적 분석 도구(dynamic analysis tool)로 나뉜다. 각각의 도구는 테스트 다이어그램 생성을 위한 정보 수집 방식과 표현 방식에서 차이를 보인다. 정적 분석 도구는 코드를 실행하지 않고도 분석할 수 있으며, 실행 흐름, 객체 간 의존성 등을 추론하는 반면, 동적 분석 도구는 실행 흐름 중에 수집된 데이터를 기반으로 분석을 수행한다.

정적 분석 도구 중 대표적인 예로는 clang-uml [7], IntelliJ IDEA Sequence Diagram plugin [8] 등이 있다. 이들 도구는 정적 코드 구조로부터 함수 간 호출 관계를 추출하여 시각적으로 표현할 수 있다. 단위 테스트 환경에서 요구되는 실행 흐름 정보를 추가로 분석할 수 있는 동적 분석 도구의 필요성이 제기된다. 그러나 이 도구는 C++의 언어적 특징을 충분히 반영하지 못하며, 단위 코드의 실행 흐름을 시각화하는 데 한계가 있다. 따라서, 단위 테스트와 단위 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

4. 결론 및 향후 연구 방향

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

본 논문은 단위 테스트 시스템에서 발생하는 단위 테스트와 실행 코드 간의 추적성 단절 문제를 해결하기 위한 방법론을 제시하고, 이를 바탕으로, 단위 테스트와 실행 코드 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다. 따라서, C++의 언어적 특징을 충분히 반영하여, 단위 테스트의 실행 흐름과 실행 코드의 실행 흐름 간의 추적성 복구를 위한 새로운 도구의 필요성이 제기된다.

출처 : KCC2025 (한국컴퓨터종합학술대회 2025)
단위 테스트-코드 간 추적성 복구를 위한 시나리오 기반 시각화 연구 - 김대원, 이영규

기존 코드

```
static int Double(int n) { return 2 * n; }

void MapTester(const Queue<int>* q) {
    const Queue<int>* const new_q = q->Map(Double);
    ASSERT_EQ(q->Size(), new_q->Size());
    for (const QueueNode<int>*n1 = q->Head(), *n2 = new_q->Head();
        n1 != nullptr; n1 = n1->next(), n2 = n2->next()) {
        EXPECT_EQ(2 * n1->element(), n2->element());
    }

    delete new_q;
}

Queue<int> q0;
Queue<int> q1;
Queue<int> q2;
```

기존 코드에 계측코드를 삽입함.

```
static int Double(int n) { trace_enter((void*)0, __PRETTY_FUNCTION__, n);
{ auto __trace_ret = 2 * n; trace_return((void*)0, __PRETTY_FUNCTION__, __trace_ret); return __trace_ret; }
}
void MapTester(const Queue<int>* q) { trace_enter((void*)this, __PRETTY_FUNCTION__, q);
const Queue<int>* const new_q = q->Map(Double);
ASSERT_EQ_LOG(q->Size(), new_q->Size());
for (const QueueNode<int>*n1 = q->Head(), *n2 = new_q->Head();
    n1 != nullptr; n1 = n1->next(), n2 = n2->next()) {
    EXPECT_EQ_LOG(2 * n1->element(), n2->element());
}

delete new_q;
}

Queue<int> q0;
Queue<int> q1;
Queue<int> q2;
```

기존 코드에 계측코드를 삽입함.

2. 계측코드가 삽입된 코드를 빌드함.

3. log를 gojs format에 맞게 변환

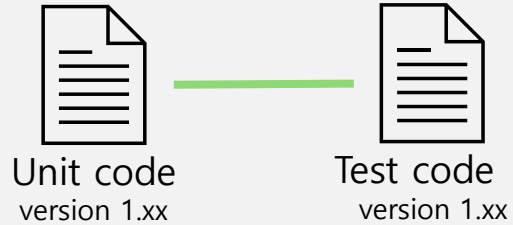
4. json파일을 gojs를 사용하여 html로 업로드

최종 시퀀스 다이어그램

TraceRecoviz를 활용한 추적성 복구

1. 기존 & 수정된 코드의 시퀀스 다이어그램 준비

추적성이 유지되던 프로젝트 상황



추적성이 끊긴 프로젝트 상황

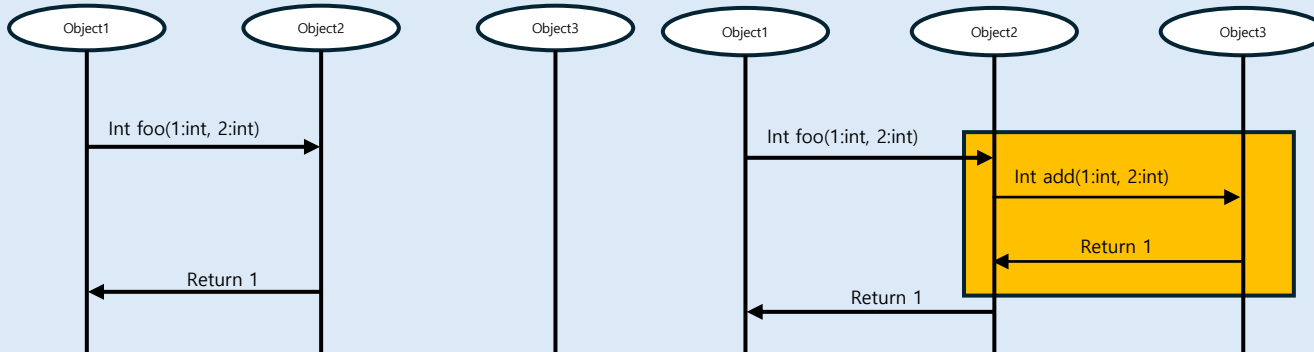
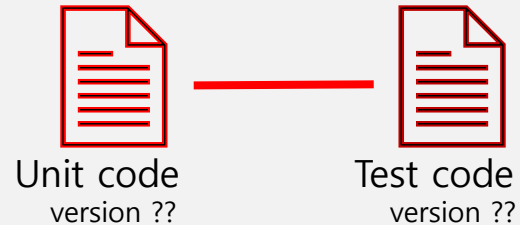


그림 1. 기존 코드의 시퀀스 다이어그램

그림 2. 수정된 코드의 시퀀스 다이어그램

2. 차이점을 비교한 시퀀스 다이어그램

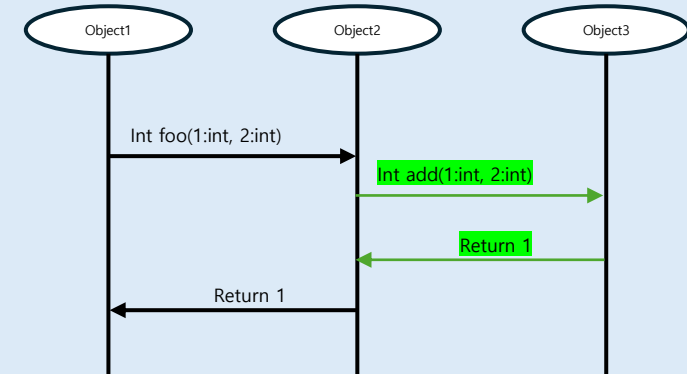
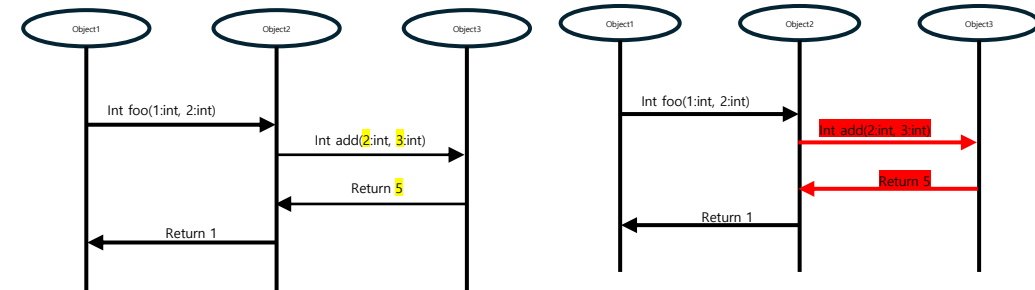


그림 3. 시퀀스 다이어그램끼리의 차이점 표시

함수 호출 추가 : 초록색
함수 입출력 변화 : 노란색
함수 호출 제거 : 빨간색



TraceRecoviz를 활용한 추적성 복구 - 새로운 함수 호출 예시

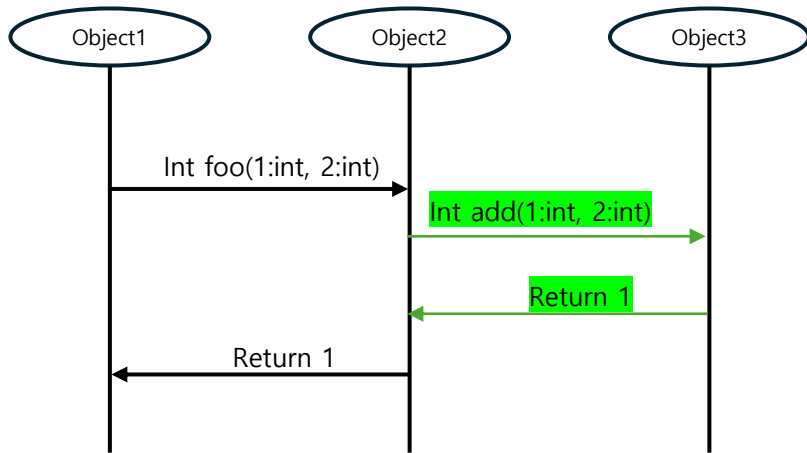


그림 1. 시퀀스 다이어그램끼리의 차이점 표시

도구가 “추적성 복구를 하는 개발자”에게 줄 수 정보

- 시퀀스 다이어그램에 초록색 표시가 생김
- 어디서 수정사항이 생겼는가? : 타겟 코드
- 어떤 요소가 수정되었는가? : 함수 내부 호출 발생
- 왜 이런 변화가 생겼는가? :
 1. 예) 동적분석 기반의 시퀀스 다이어그램에 표시되지 않는 요소의 수정
 - if 문 조건의 변화
 - while, for 같은 반복문의 변화
 2. 개발자가 새로운 호출을 추가함
 3. ~~
- 무엇을 해야하는가? :

함수 내부의 호출이 생긴 foo() 내부를 한번 확인해봐야 합니다.
새로운 함수호출이 생긴 부분이 출력값에 어떤 영향을 주는지 확인해봐야합니다

TraceRecoviz를 활용한 추적성 복구 - GTEST Sample1 예시

1. 추적성이 유지되는 상황

Version 1.1

```
int Factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result *= i;  
    }  
  
    return result;  
}
```

그림 1. 타겟 코드 - 반복문으로 작성된 팩토리얼 함수

```
TEST(FactorialTest, Negative) {  
    EXPECT_EQ(1, Factorial(-5));  
    EXPECT_EQ(1, Factorial(-1));  
    EXPECT_GT(Factorial(-10), 0);  
}  
// Tests factorial of 0.  
TEST(FactorialTest, Zero) { EXPECT_EQ(1, Factorial(0)); }  
// Tests factorial of positive numbers.  
TEST(FactorialTest, Positive) {  
    EXPECT_EQ(1, Factorial(1));  
    EXPECT_EQ(2, Factorial(2));  
    EXPECT_EQ(6, Factorial(3));  
    EXPECT_EQ(40320, Factorial(8));  
}
```

그림 2. 테스트 코드

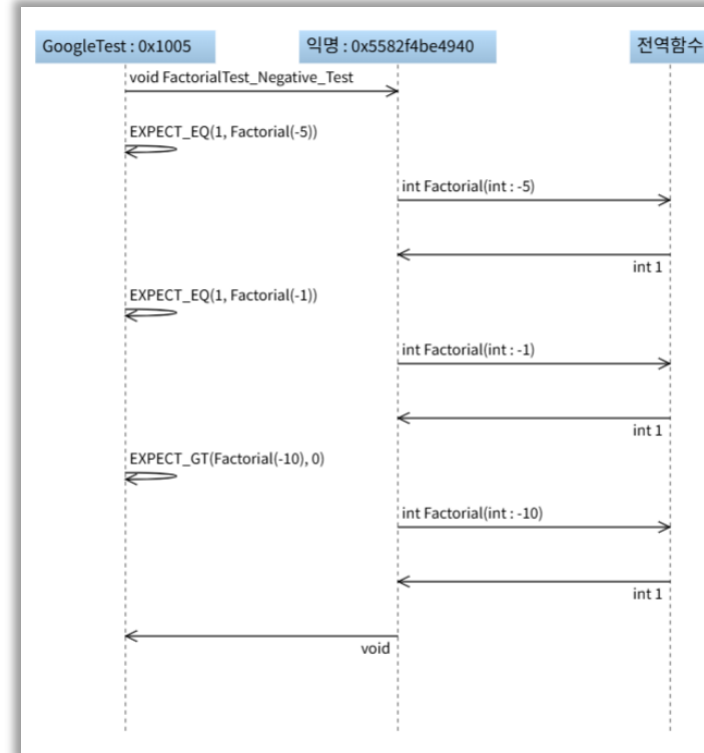


그림 4. 반복문으로 구현된 팩토리얼 함수의 테스트 코드의 시퀀스 다이어그램

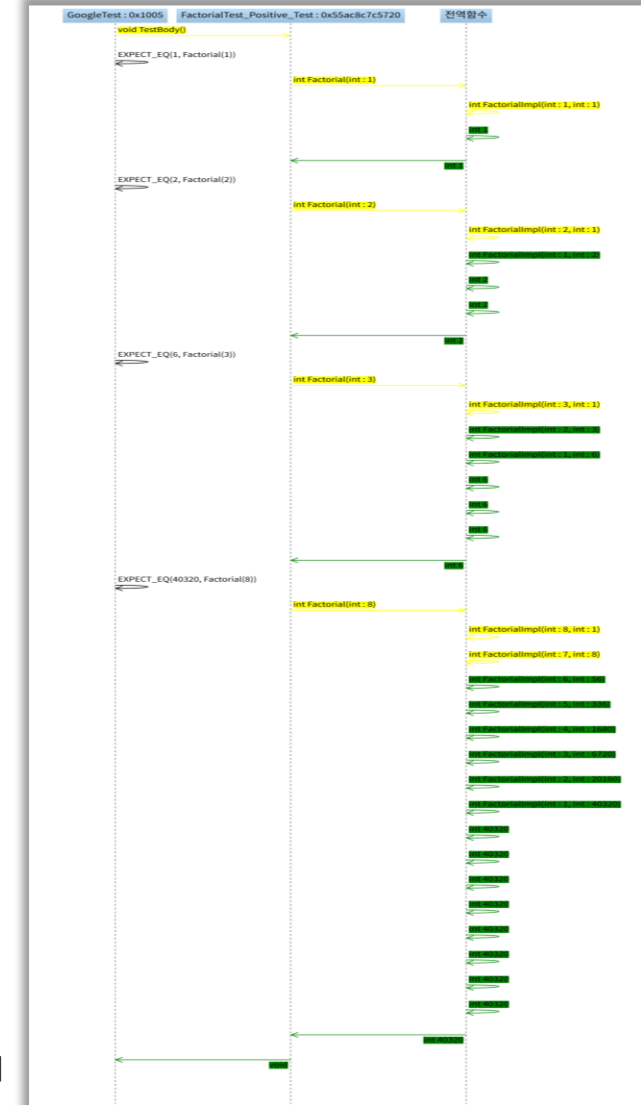


그림 5. 재귀로 구현된 팩토리얼 함수의 테스트 코드의 시퀀스 다이어그램

2. 추적성이 끊긴 상황

Version 1.2

```
int FactorialImpl(int k, int acc) {  
    if (k <= 1) return acc;  
    return FactorialImpl(k - 1, acc * k);  
}  
  
int Factorial(int n) {  
    return FactorialImpl(n, 1);  
}
```

그림 3. 수정 후 타겟 코드 - 재귀로 구현된 팩토리얼 함수

TraceRecoviz를 활용한 추적성 복구 – GTEST Sample1 예시

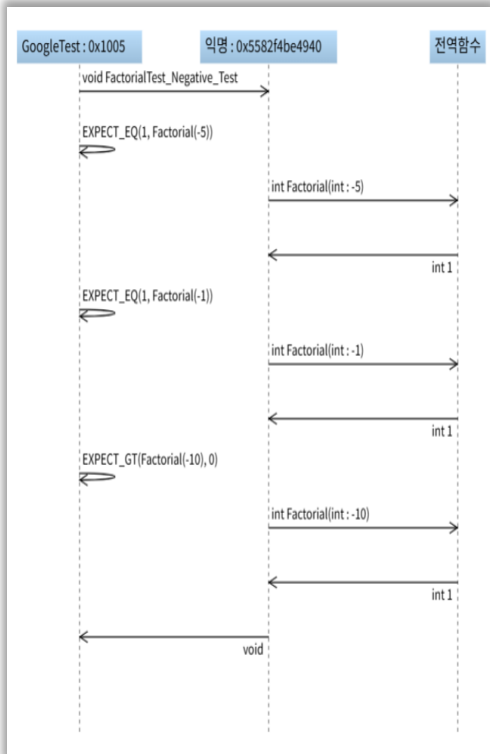


그림 4. 반복문으로 구현된 팩토리얼 함수의 테스트 코드의 시퀀스 다이어그램

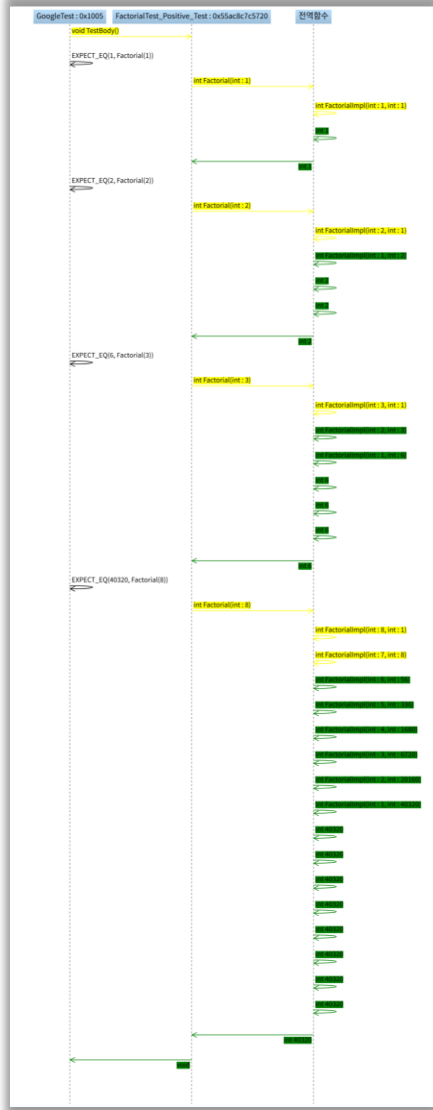


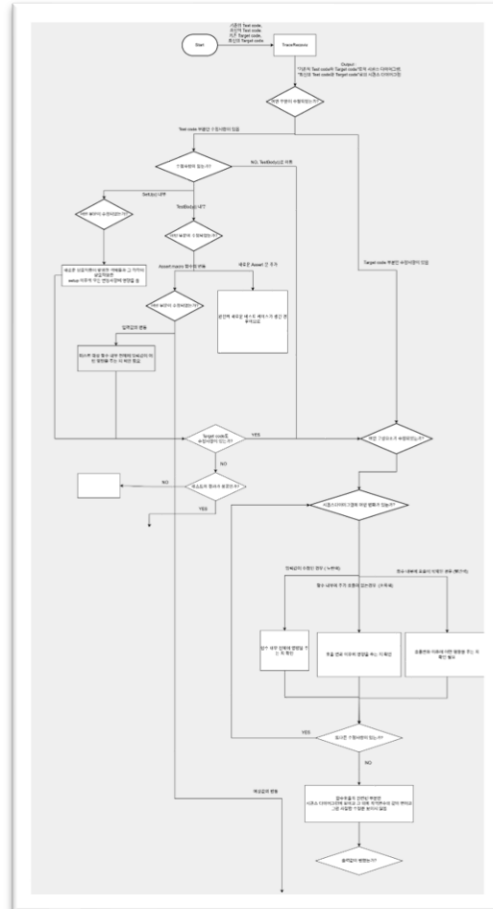
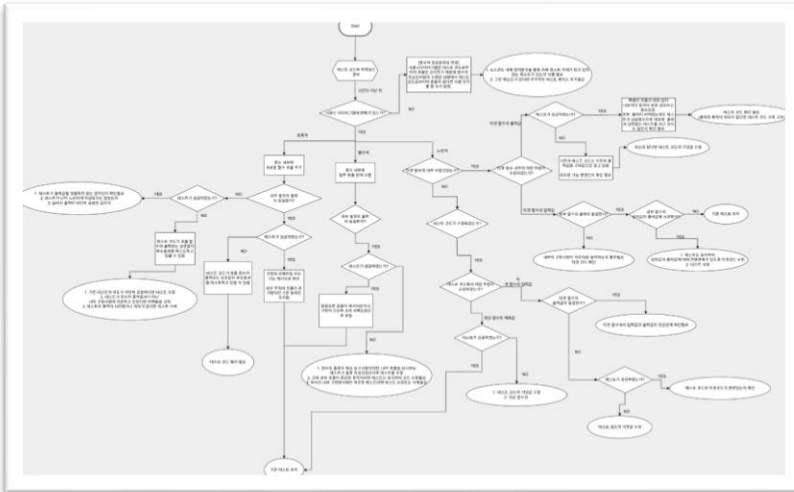
그림 5. 재귀로 구현된 팩토리얼 함수의 테스트 코드의 시퀀스 다이어그램

도구가 “추적성 복구를 하는 개발자”에게 줄 수 정보

- 시퀀스 다이어그램에 초록색 표시가 생김
- 어디서 수정사항이 생겼는가? : 타겟 코드
- 어떤 요소가 수정되었는가? : 함수 내부 호출 발생
- 왜 이런 변화가 생겼는가? :
 1. 예) 동적분석 기반의 시퀀스 다이어그램에 표시되지 않는 요소의 수정
 - if 문 조건의 변화
 - while, for 같은 반복문의 변화
 2. 개발자가 새로운 호출을 추가함
 3. Caller 함수와 Callee 함수 시그니처가 동일함
 - > 재귀함수 호출로 코드를 수정
- 무엇을 해야하는가? :

Caller 함수와 내부의 Callee 함수 시그니처가 동일한 것으로 보아 재귀함수로 코드를 수정한 것으로 보입니다.
Caller 함수 내부를 한번 확인해봐야 합니다.

TraceRecoviz를 활용한 추적성 복구 - 발생가능한 케이스 정리



시나리오 (변경 유형)	시퀀스 다이어그램 변화 및 테스트 영향	원인 (코드/테스트 수정)	추적성 복구 개발자 대응
1. 함수 호출 추가 (타겟 코드 내부)	내부에 새로운 함수 호출 메시지가 다이어그램에 추가됨. 출력 등 외부행동 동일하면 테스트 성공 (테스트 코드 변경 없음).	구현 리팩토링 또는 기능 개선으로 내부 로직에 호출 추가.	테스트 유지 (수정 불필요). 다이어그램 변화는 이력으로 기록.
2. 함수 호출 삭제 (타겟 코드 내부)	기존 다이어그램의 일부 호출 메시지 소멸. 외부행동 동일하면 테스트 성공 (테스트 코드 변경 없음). ※ 기대된 호출 없다면 mock 테스트 실패 가능.	불필요한 호출 제거, 구현 단순화 등의 리팩토링. 또는 기능 축소/제거.	외부동작 불변 시 테스트 유지. 기능 제거 시 해당 테스트 기대치 수정 또는 테스트 폐기.
3. 함수 호출 변경 (호출 대상/인자 변경)	다이어그램에 나타나는 호출의 대상 함수명이나 인자 형태 변화. 출력이 변하지 않으면 테스트 성공 (코드 호출부만 간접 영향, 일반적으로 테스트 코드 그대로).	효율 개선 위해 호출 대상 교체 (예: 커스텀 - > 표준 라이브러리) 또는 API 변경.	기능동일시 테스트 유지. 출력에 영향 시 6번 대응처럼 예상값 수정. 추적성 정보 업데이트.
4. 함수 시그니처 변경 (인터페이스 변경)	다이어그램에 호출 자체가 변경됨 (함수명/파라미터 달라짐). 테스트는 컴파일/실행 불가로 실패. (기존 테스트의 assert도 맞지 않을 수 있음)	함수 프로토타입 수정: 이름 변경, 파라미터 추가/변경, 반환형 변경 등 (주로 요구사항 변경에 따른 API 수정).	테스트 코드 수정하여 새로운 시그니처 호출하도록 보완. 대규모 변경 시 기존 테스트 폐기 후 신규 테스트 작성 고려.
5. 입력값 처리 변경 (입력 제약/검증 수정)	다이어그램에 예외 흐름 추가 또는 분기 변화. 특정 입력에 대한 동작 변경으로 해당 케이스 테스트 실패 (assert 예상값 불일치 혹은 예외 미처리).	입력값에 대한 처리 로직 변경 (예: 유효성 검사 추가, 범위 제한, 예외 처리 방식 변경) - 요구사항 수정이나 버그 해결.	해당 케이스의 테스트 수정: 예상 결과를 새 규칙에 맞게 변경하거나 예외 체크 추가. 불필요해진 테스트는 제거.
6. 출력값/동작 변경 (반환값 또는 효과 수정)	다이어그램 구조는 동일하나 반환값 등 결과 달라짐. 테스트 실패 (기대된 출력과 불일치).	함수의 결과값 정의의 변경 또는 부수 효과 수정 - 요구사항 변경이나 설계 개선.	테스트 Orade 갱신: EXPECT 등의 예상값을 변경. 기능 의미 변화 크면 테스트 전면 개편 또는 신규 작성.
7. 테스트 케이스 추가 (새 시나리오 테스트)	기존 다이어그램은 영향 없음. 새로운 테스트에 대한 시퀀스 다이어그램 추가 생성. 전체 테스트 스위트에 새 실행 경로가 늘어남.	코드의 기능 확장이나 신규 요구사항으로 추가된 케이스. 또는 테스트 커버리지 향상 목적의 추가.	기존 테스트 유지. 새 테스트를 스위트에 포함하고 추적성 매트릭스에 신규 항목 추가.
8. 기능 삭제/대체 (대상 기능 자체 제거)	관련 시퀀스 다이어그램이 사라짐 (호출 자체가 없어짐). 해당 테스트는 의미 없어져 실패 또는 스킵.	요구사항 폐기, 아키텍처 변경 등으로 기능 제거 또는 다른 구현으로 교체.	관련 테스트 폐기 (더 이상 사용하지 않음). 대체 기능이 있으면 그에 맞는 새로운 테스트 작성. 추적성 링크 정리/갱신.

TraceRecoviz를 활용한 추적성 복구 - 검증

- 기존의 작업된 프로젝트를 직접 사용하여 실사례로 추적성복구 시나리오 테스트
- 타겟 코드, 테스트 코드, 수정된 코드(타겟 코드, 테스트 코드)