unit test의 시나리오 기반 시각화

1. 프로젝트 개요

문제 정의
- **레거시 시스템**에서는 문서화 부족으로 인해 unit test code가 개발자의 의도대로 동작하는지 판단하기 어려움
- 단순히 unit test를 통과했다 하더라도, 실제 시스템이 기대한 대로 동작한다고 단정할 수 없음

개선 필요성
- 이러한 문제를 해결하려면 코드의 동작 과정, 입력값, 기댓값이 무엇인지 명확히 이해할 수 있어야 함. 즉, 특정 입력값이 주어졌을 때 코드가 어떤 절차를 거쳐 실행되며, 최종적으로 기대하는 출력값이 무엇인지 확인할 수 있어야 함
- 레거시 시스템에서는 코드가 복잡하고 문서화가 부족하여 개발자가 직접 읽고 분석하는 데 많은 시간이 소요됨. 따라서, 흩어진 코드의 흐름을 한눈에 볼 수 있는 방법이 필요함

제안하는 해결책
- 테스트 시나리오는 특정 기능이나 요구사항이 예상대로 동작하는지 검증하기 위한 일련의 절차로, 입력값, 동작과정, 기댓값이 기술되어 있어서 개발자의 의도대로 동작하는지 확인하는데 도움이 됨
- **시퀀스 다이어그램**은 시스템 내 객체 간의 상호작용을 시간 순서에 따라 시각적으로 표현하는 도구로, unit test의 실행 흐름을 한눈에 파악하는 데 도움을 줌
- 따라서, **테스트 시나리오를 기반으로 하는 시퀀스 다이어그램**을 그리면 unit test code가 개발자의 의도대로 동작하는지 파악하는데 도움을 줌

결론
- 그러나, 매번 수작업으로 시각화하는 것은 비용 및 시간 측면에서 비효율적임
- 또한, 기존 도구는 단순히 코드의 구조나 호출관계를 시각화하는 데에만 초점이 맞춰져 있음
- 따라서, 기존 도구를 테스트 시나리오를 기반으로 시퀀스 다이어그램을 **자동으로 생성하는 도구로 확장**하고자 함

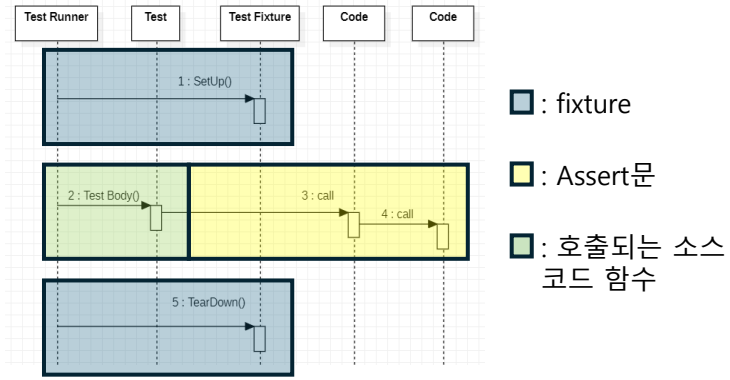
2. 테스트 시나리오 기반 시퀀스 다이어그램

- 본 연구의 대상을 C++ 테스트 프레임워크인 Google Test의 Sample Code로 선정함
- 테스트 코드 기반 시퀀스 다이어그램에 들어갈 요소와 테스트 시나리오의 구조를 표현함 [표1, 그림1]

표1. 시퀀스 다이어그램에 들어가야 할 Google Test 요소

Test 구성요소	역할
Fixture	테스트에 필요한 데이터나 환경을 준비
Stub	특정 기능을 dummy 객체로 대체하는 부분을 표시
Assert 문	입력값과 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	이외에 테스트 프레임워크에서 사용되는 객체들

그림1. Google Test에 기반한 기본적인 테스트 구조



3. 기존 도구의 분석

기존 도구들을 분석해본 결과, 테스트 시나리오를 이해하는데 도움을 주기에는 부족함 [표2]

기존 도구의 한계
- 기존 시퀀스 다이어그램에서 테스트 코드에 대한 정보가 부족하거나, 부정확함
- **생략된 defualt 생성자, 소멸자는 표현이 안돼서** 객체의 생성과 소멸을 이해하기에 정보가 부족함
- **overloading된 new 와 delete**: 시퀀스 다이어그램에 호출하는 위치가 정확하지 않음
- **TEST_F() 매크로 함수**에서는 테스트객체를 제대로 표현하지 못함
- **Assert문**에 명시된 인자와 기대 값을 표시하지 못함
- **테스트 프레임워크에서 사용하는 객체**가 언제 어떻게 호출되었는지 표시되지 않음
- **업캐스팅**이 발생한 경우 실제 동작하는 자식객체가 아닌 부모객체로 표현됨

표2. 기존 도구 비교

	SequenceDiagram for C++(JetBrains Plugin)	UModel (Altova)	clang-uml
분석 방식	정적	정적	정적
메서드 호출 흐름	O	X	O
조건문 및 분기 구조 표시	X	O	O
오픈소스	X	X	O
Assert문 표시	X	X	X
객체 라이프 사이클 표시	X	X	X
Google Test 구조 표현	X	X	X

4. 해결 방안
clang-uml을 수정*확장한 도구 개발

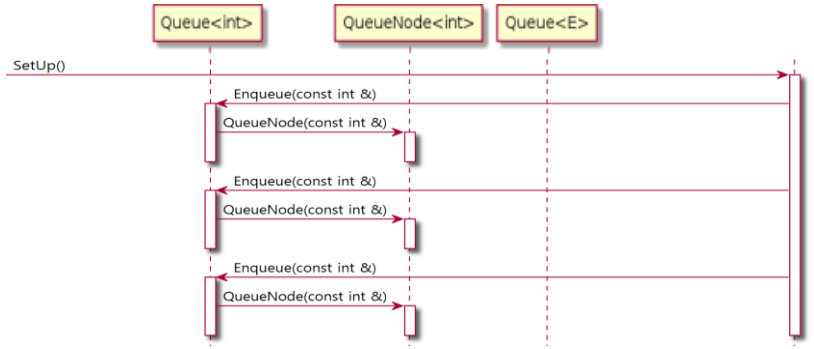
clang-uml는 오픈소스이므로 소스코드를 활용해서 테스트 코드 기반 시퀀스
다이어그램을 그리는 도구를 개발하고자 함

기존 시퀀스 다이어그램에서 테스트 시나리오에 대한 정보가 부족함

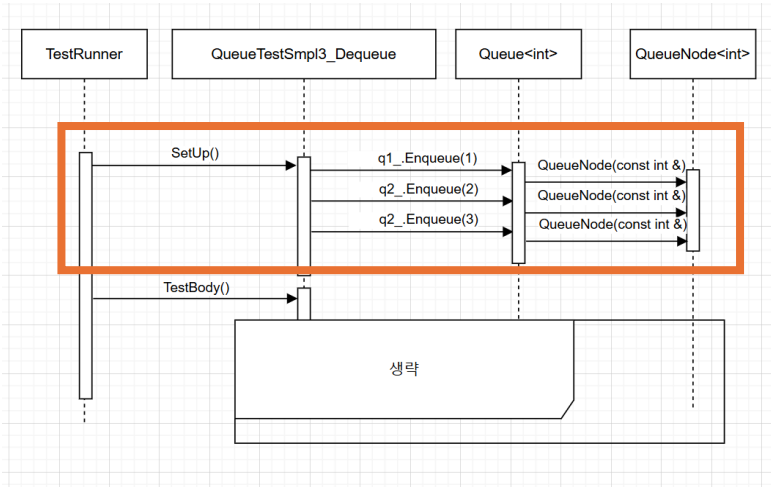
- 코드 상에는 q1_.Enqueue가 1번, q2_.Enqueue가 2번 호출되도록 작성되어 있음 **[1]**
- 하지만, 기존 도구로 생성된 시퀀스 다이어그램에서는 단순히 Enqueue가 3번 호출되어 있어서 테스트 코드를 이해하기가 어려움 **[2]**
- 따라서, 그림처럼 호출객체를 명시해줌으로써 테스트 시나리오를 이해하는데 도움을 주고자 함 **[3]**
- **객체 기반 시퀀스 다이어그램으로 그리면 객체의 흐름을 보다 더 명확하게 알 수 있지만 정적분석으로 어느정도까지 가능한지 확인 필요함[4, 5]**

```
class QueueTestSmpl3 : public testing::Test {
 protected:
  void SetUp() override {
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }
```

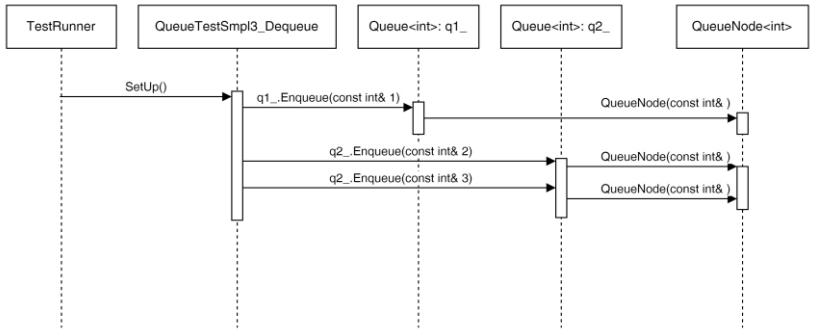
[1] Sample3. 테스트코드



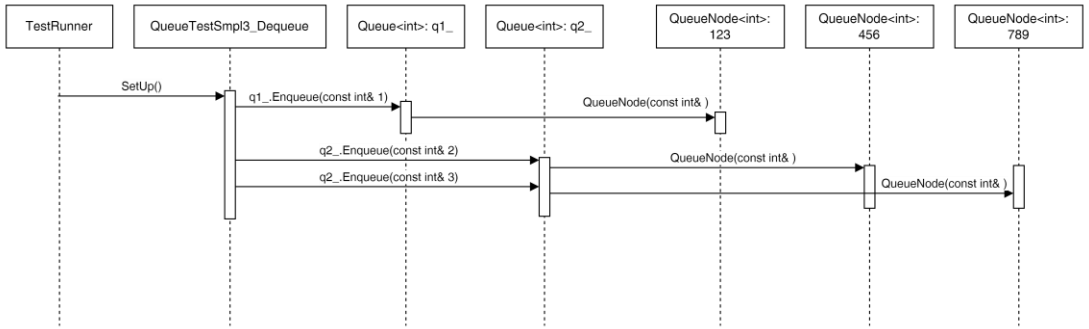
[2] Sample3. clang-uml에서 그려준 그림



[3] Sample3. 원하는 그림



[4] 객체 수준의 시퀀스 다이어그램 – depth 1



[5] 객체 수준의 시퀀스 다이어그램 – depth n

생략된 default 생성자, 소멸자는 표현이 되지 않아서 객체의 생성과 소멸을 이해하기에 정보가 부족함

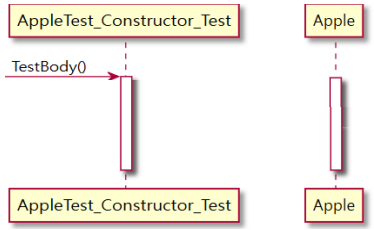
- C++에서는 기본 생성자와 소멸자가 명시적으로 선언되지 않더라도 자동으로 제공됨 **[1, 2, 3, 4]**
- 기존의 다이어그램 생성 도구에서는 코드에서 이를 생략하면 다이어그램에도 포함되지 않는 문제가 발생함 **[5, 6]**
- 이러한 문제로 인해 다이어그램이 코드의 실제 동작을 반영하지 못하며, 객체의 생성과 소멸 과정이 명확하게 표현되지 않음
- 따라서, 다이어그램의 정확성을 확보하기 위해 기본 생성자와 소멸자가 코드에서 생략되었더라도 이를 다이어그램에 포함하는 방식으로 구현해야 함 **[7]**

```
class Apple {
 public:
  // Normal Apple declarations go here.
};
```

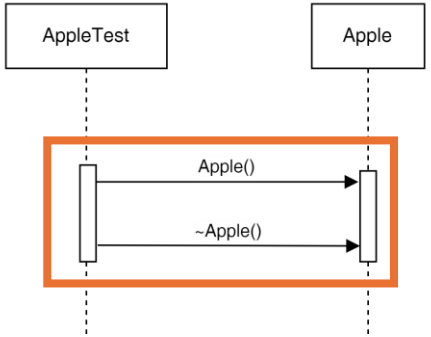
[1] Sample11. 디폴트 생성자가 생략된 코드

```
TEST(AppleTest, constructor) {
    Apple water1 = Apple();
    EXPECT_NE(&water1, nullptr);
}
```

[2] Sample11. 디폴트 생성자가 생략된 테스트 코드



[5] Sample11. 디폴트 생성자가 생략된 코드를 clang-uml에서 그려준 그림



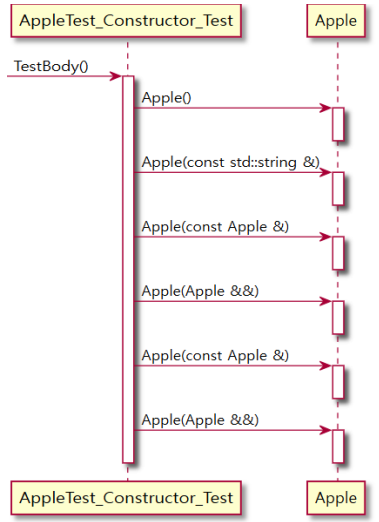
[7] 그림 5를 개선한 그림

```
class Apple {
 public:
    Apple() {};
    Apple(const std::string& color):color_(color) {};
    Apple(const Apple& other):color_(other.color_) {};
    Apple(Apple&&other) noexcept : color_(std::move(other.color_)){};
    ~Apple() {};
    Apple& operator=(const Apple& other) {
        if (this != &other) {
            color_ = other.color_;
            // Do the assignment
        }
        return *this;
    }
    Apple& operator=(Apple&& other) noexcept {
        if (this != &other) {
            color_ = std::move(other.color_);
            // Do the assignment
        }
        return *this;
    }
protected:
    std::string color_;
  // Normal Apple declarations go here.
};
```

[3] Sample11. 다양한 생성자가 명시된 코드

```
TEST(AppleTest, Constructor) {
    Apple a1;
    Apple a2("green");
    Apple a3(a2);
    Apple a4(std::move(a3));
    Apple a5 = a4;
    Apple a6 = std::move(a5);

}
```

[4] Sample11. 다양한 생성자가 명시된 테스트 코드



[6] Sample11. 다양한 생성자가 명시된 코드를 clang-uml에서 그려준 그림

재정의된 new 와 delete는 시퀀스 다이어그램에서 표시 해줘야 함 (1/2)

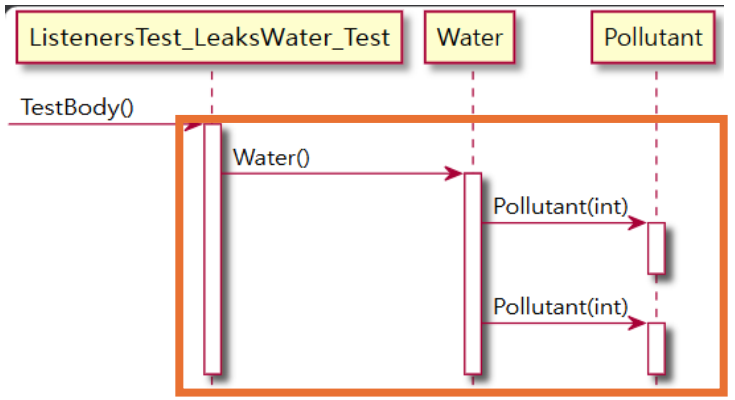
- 코드에서 Water는 operator new와 operator delete가 재정의되어 있음 **[1, 2]**
- 도구를 통해 생성된 시퀀스 다이어그램에서 이 부분이 표현되지 않아, 재정의된 연산자가 사용되었는지 확인할 수 없는 문제가 발생함 **[3]**
- 이를 해결하기 위해 시퀀스 다이어그램에서 재정의된 new 및 delete 호출이 명확하게 나타나도록 수정해야 함 **[4]**

```
class Pollutant {
  public:
    Pollutant();
    Pollutant(int concentration);
    ~Pollutant();

    int concentration;
};


/ We will track memory used by this class.
lass Water {
public:
 Water() {
   Pollutant* pollutant1 = new Pollutant(1);
   Pollutant* pollutant2 = new Pollutant(10);
 }
 void* operator new(size_t allocation_size) {
   allocated_++;
   Pollutant* pollutant1 = new Pollutant();

   return malloc(allocation_size);
 }

 void operator delete(void* block, size_t /* allocation_size */) {
   allocated_--;
   //Water *a = new Water();
   free(block);
 }

 static int allocated() { return allocated_; }

rivate:
 static int allocated_;
};
```

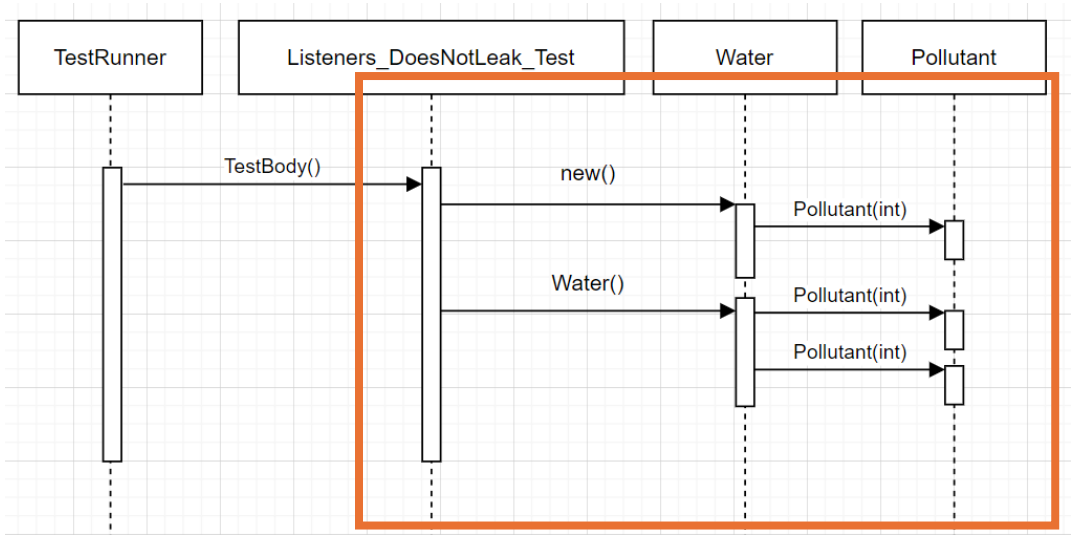
[1] Sample10. operator가 재정의된 코드

```
TEST(ListenersTest, DoesNotLeak) {
  Water* water = new Water;
  delete water;
}
```

[2] Sample10. 테스트 코드



[3] Sample10. clang-uml에서 그려준 그림



[4] Sample10. 원하는 그림

재정의된 new 와 delete는 시퀀스 다이어그램에서 표시 해줘야 함 (2/2)

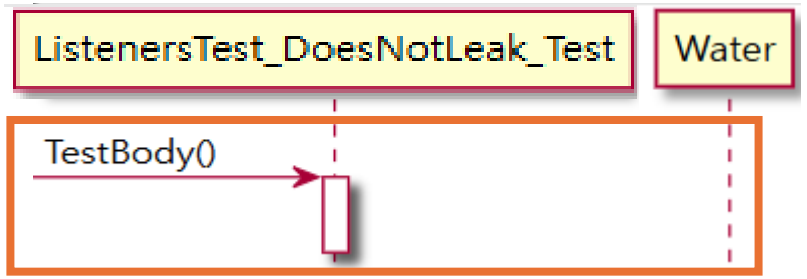
- 코드에서 Water는 operator new와 operator delete가 재정의되어 있음 **[1, 2]**
- 도구를 통해 생성된 시퀀스 다이어그램에서 이 부분이 표현되지 않아, 재정의된 연산자가 사용되었는지 확인할 수 없는 문제가 발생함 **[3]**
- 이를 해결하기 위해 시퀀스 다이어그램에서 재정의된 new 및 delete 호출이 명확하게 나타나도록 수정해야 함 **[4]**
- **생성자도 없고, 재정의된 new도 없고**

```
class Water {
 public:
  // Normal Water declarations go here.
  // operator new and operator delete help us cont
  void* operator new(size_t allocation_size) {
    allocated_++;
    //Water *a = new Water();
    return malloc(allocation_size);
  }

  void operator delete(void* block, size_t /* all
    allocated_--;
    //Water *a = new Water();
    free(block);
  }

  static int allocated() { return allocated_; }

 private:
  static int allocated_;
};
```

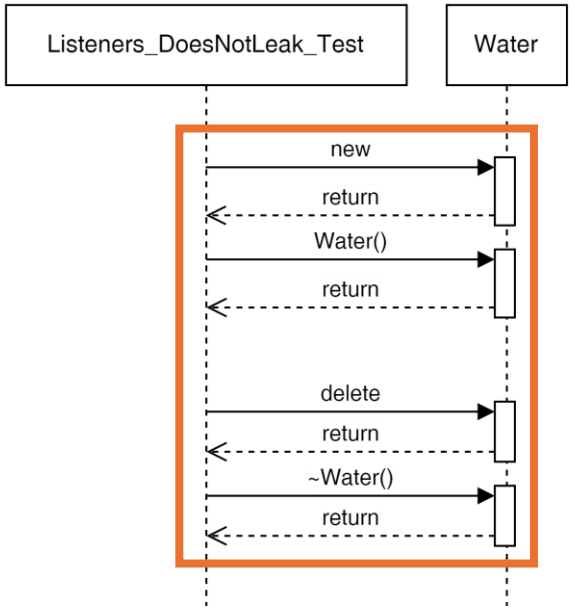
[1] Sample10. overloading된 코드

```
TEST(ListenersTest, DoesNotLeak) {
  Water* water = new Water;
  delete water;
}
```

[2] Sample10. 테스트 코드



[3] Sample10. clang-uml에서 그려준 그림



[4] Sample10. 원하는 그림

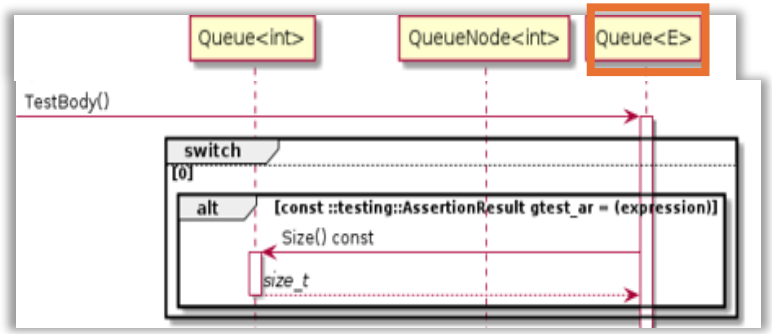
TEST_F() 매크로 함수에서는 테스트객체를 제대로 표현하지 못함.

- TEST_F() 매크로 함수에서 TestBody()의 호출대상 객체**[1]**가 Queue<E>로 잘못 표시되는데 QueueTestSmpl3_DefaultConstructor로 수정이 필요함 **[2, 3]**
- TSET_P() 매크로 함수에서는 테스트 객체를 정확하게 명시하고 있음 **[4, 5]**

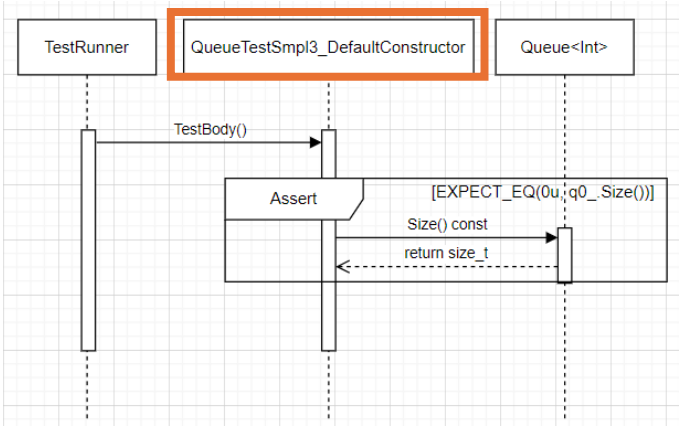
Sample 3

```
TEST_F(QueueTestSmpl3, DefaultConstructor) {
  // You can access data in the test fixture
  EXPECT_EQ(0u, q0_.Size());
}
```

[1] Sample3. TEST_F 테스트코드



[2] Sample3. clang-uml에서 그려준 그림

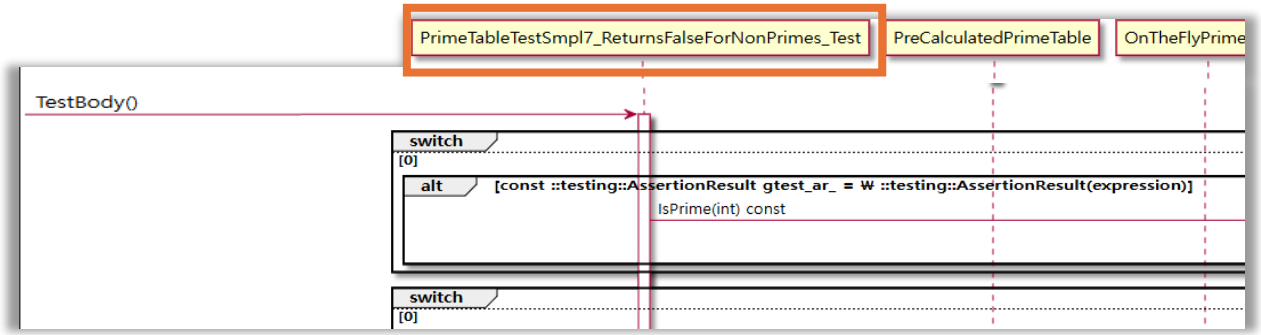


[3] Sample3. 원하는 그림

Sample 7

```
TEST_P(PrimeTableTestSmpl7, ReturnsFalseForNonPrimes) {
  EXPECT_FALSE(table_->IsPrime(-5));   // primetable 에
  EXPECT_FALSE(table_->IsPrime(0));
  EXPECT_FALSE(table_->IsPrime(1));
  EXPECT_FALSE(table_->IsPrime(4));
  EXPECT_FALSE(table_->IsPrime(6));
  EXPECT_FALSE(table_->IsPrime(100));
}
```

[4] Sample7. TEST_F 테스트코드



[5] Sample7. clang-uml에서 그려준 그림

Assert문에 명시된 인자와 기댓값을 표시 못함

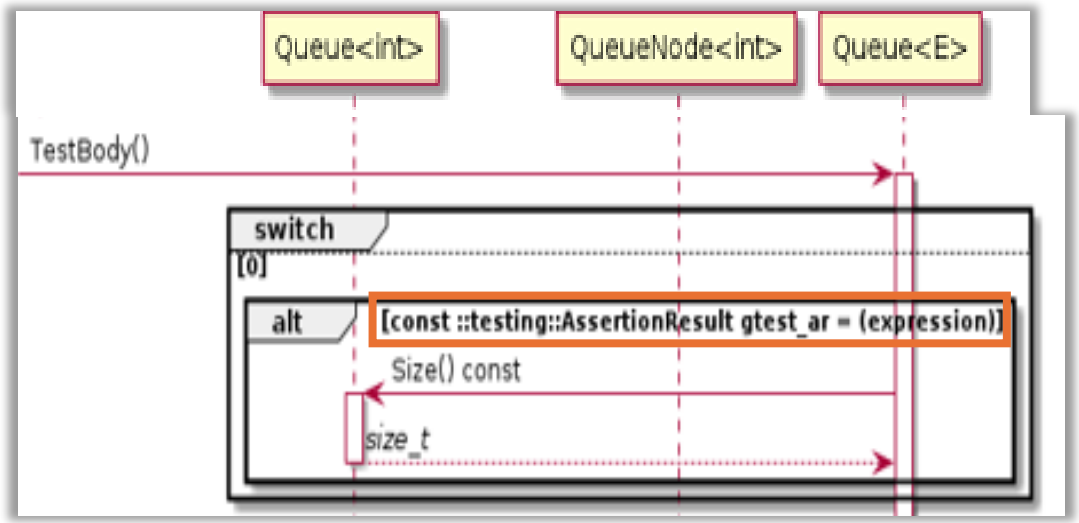
- clang-uml에서는 EXPECT_EQ()같은 Assert문**[1]**을 const::testing::AssertionResult gtest_ar = (expression) 으로 표시함 **[2, 3]**
- 무엇을 테스트하는 함수인지 바로 알기 어려움
- 테스트코드에 명시된대로 EXPECT_EQ()로 수정필요 **[1, 3]**

```
TEST_F(QueueTestSmpl3, DefaultConstructor) {
  // You can access data in the test fixture
  EXPECT_EQ(0u, q0_.Size());
}
```

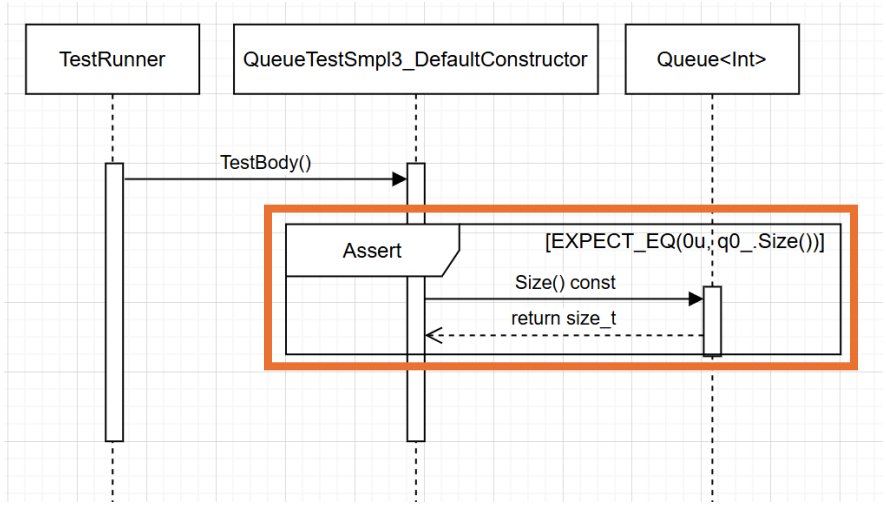
[1] Sample3. Asset문 테스트코드

```
#define GTEST_ASSERT_(expression, on_failure)
  GTEST_AMBIGUOUS_ELSE_BLOCKER_
  if (const ::testing::AssertionResult gtest_ar = (expression))
    ;
  else
    on_failure(gtest_ar.failure_message())
```

[2] Sample3. Asset문의 정의



[3] Sample3. clang-uml에서 그려준 그림



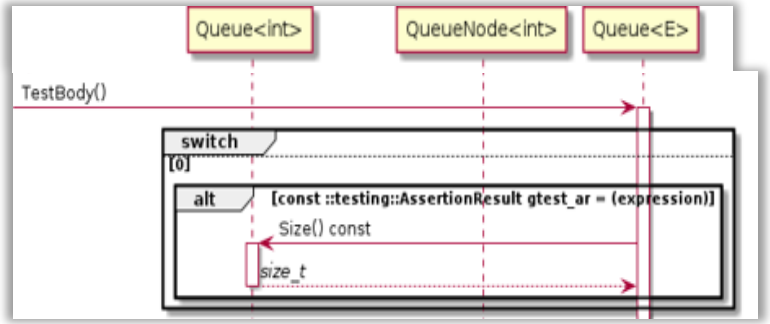
[4] Sample3. 원하는 그림

테스트 프레임워크에 사용하는 객체가 언제 어떻게 호출되었는지 표시가 안됨

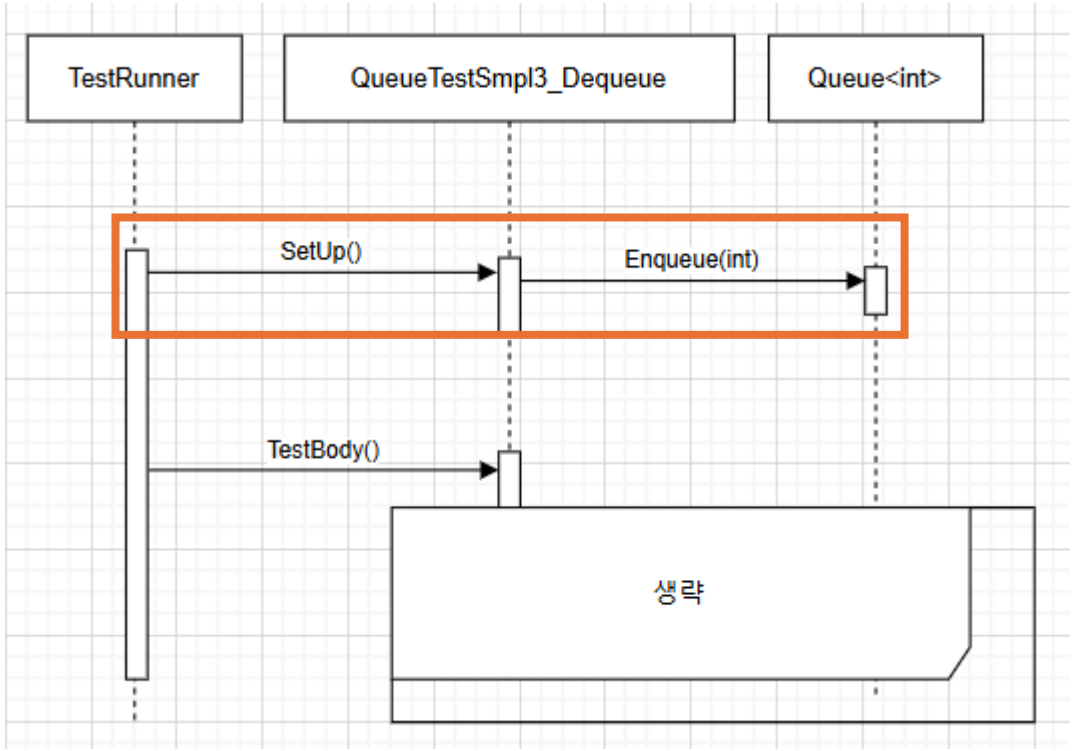
- Setup 에서 공통적인 Test Suite 환경을 설정하므로 이에 대한 정보가 있을 시 테스트 시나리오를 이해하는데 도움이 됨
- **[1]**에서 setup 부분에서 q1,q2에 enqueue()를 했다는 정보를 알아야 하지만, **[2]**에서는 이러한 정보가 보이지 않음
- Setup은 TestBody 호출 전에 표시가 되어야 하고, TearDown은 TestBody가 호출이 끝난 후에 표시가 되어야 함 **[3]**

```
class QueueTestSmpl3 : public testing::Test {
 protected:
  void SetUp() override {
    q1_.Enqueue(1);
  }
```

[1] Sample3. 테스트코드



[2] Sample3. clang-uml에서 그려준 그림



[3] Sample3. 원하는 그림

업캐스팅이 발생한 경우, 실제 동작하는 자식객체가 아닌 부모객체로 표현됨

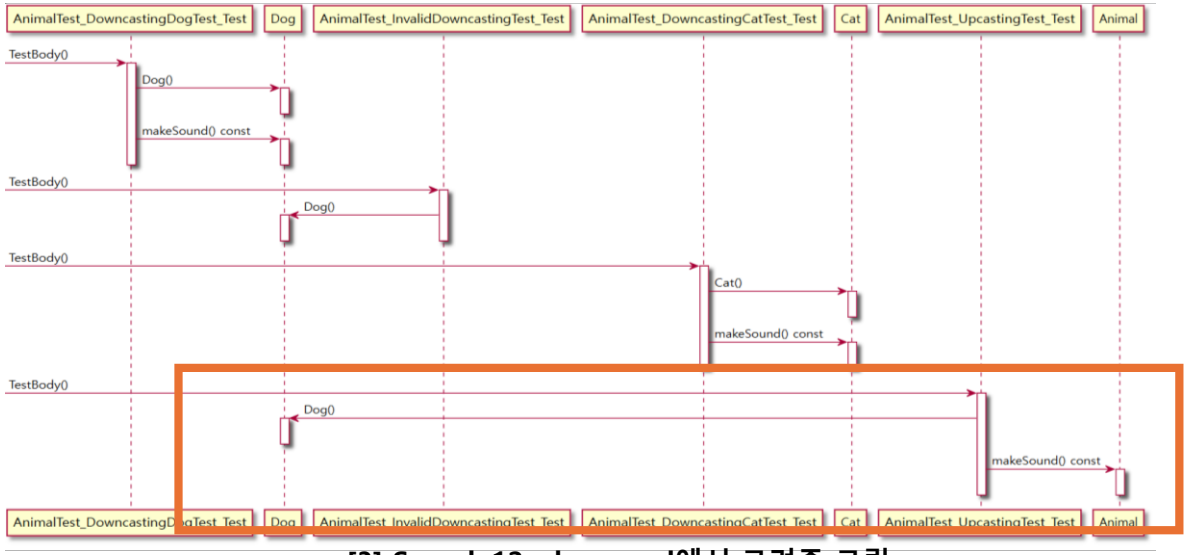
- 자식 클래스 타입인 myDog 변수를 업캐스팅하여 부모 클래스 타입인 animal 변수로 선언하였음 **[1, 2]**
- **animal 변수가 makeSound를 호출하면 clang-uml에서도 그대로 그려줌 [3]**
- 그러나 실제 실행되는 메소드는 자식 클래스에서 재정의된 메소드이므로, 해당 시퀀스 다이어그램을 보고는 실행 흐름을 정확히 이해하기 어려움
- 이를 해결하기 위해 업캐스팅이 적용되었더라도 실제 실행되는 메소드 호출이 올바르게 표현되도록 부모객체가 아닌 자식객체로 표현되어야 함 **[4]**

```cpp
class Animal {
public:
    Animal();
    Animal(const std::string& name);
    virtual ~Animal();
    virtual void makeSound() const = 0;
    virtual void displayInfo() const;

protected:
    std::string name_;
};

class Dog : public Animal {
public:
    Dog();
    Dog(const std::string& name);

    ~Dog();

    void makeSound() const override;
    void fetch() const;
};

class Cat : public Animal {
public:
    Cat();
    Cat(const std::string& name);

    ~Cat();

    void makeSound() const override;
    void climb() const;
};
```

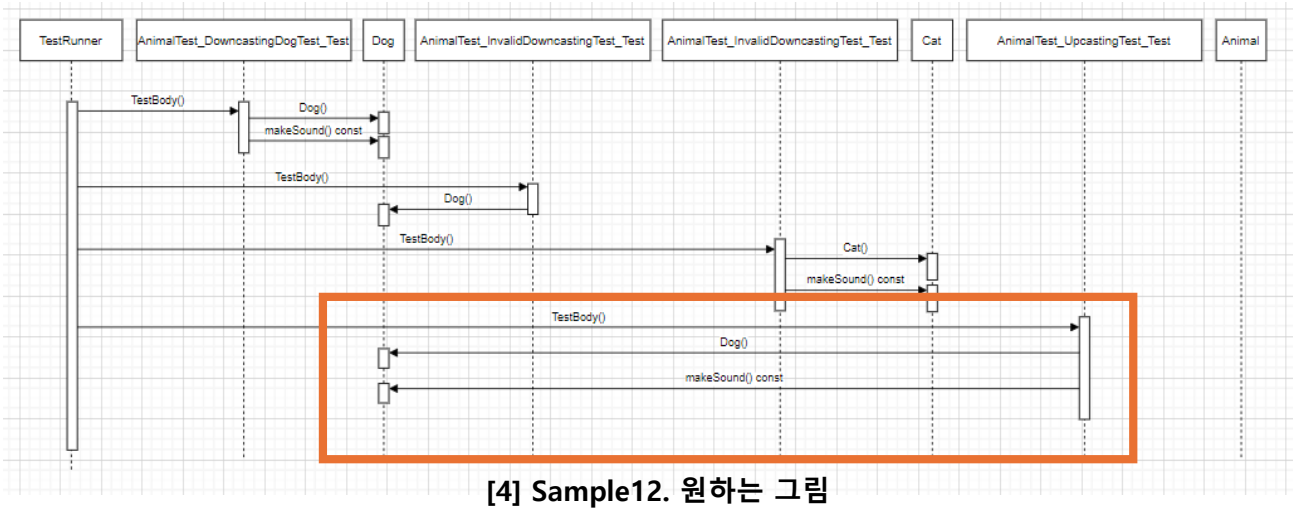
[1] Sample12. Animal Class

```cpp
// 업캐스팅 테스트
TEST(AnimalTest, UpcastingTest) {
    Dog myDog;
    Animal* animal = &myDog;
    animal->makeSound();
}

// 다운캐스팅 테스트 - Dog
TEST(AnimalTest, DowncastingDogTest) {
    Dog myDog;
    Animal* animal = &myDog;

    Dog* dog = dynamic_cast<Dog*>(animal);
    dog->makeSound();
}

// 다운캐스팅 테스트 - Cat
TEST(AnimalTest, DowncastingCatTest) {
    Cat myCat;
    Animal* animal = &myCat;

    Cat* cat = dynamic_cast<Cat*>(animal);
    cat->makeSound();
}

// 잘못된 다운캐스팅 테스트
TEST(AnimalTest, InvalidDowncastingTest) {
    Dog myDog;
    Animal* animal = &myDog;

    Cat* cat = dynamic_cast<Cat*>(animal);
}
```

[2] Sample12. Test Code



[3] Sample12. clang-uml에서 그려준 그림



[4] Sample12. 원하는 그림