Design Patterns in C++
https://refactoring.guru/design-patterns/cpp

1. Abstract Factory

- Google Test 라이브러리에서 쓰임

- unit testing 예제:
llvm/unittests/ADT/APIntTest.cpp

아래 코드에서 TEST는 매크로로써, Test의 자식 클래스를 정의합니다.
Test class의 정의: llvm/utils/unittest/googletest/include/gtest/gtest.h

```
// Test that APInt shift left works when bitwidth > 64 and shiftamt == 0
TEST(APIntTest, ShiftLeftByZero) {
 APInt One = APInt::getNullValue(65) + 1;
 APInt Shl = One.shl(0);
 EXPECT_TRUE(Shl[0]);
 EXPECT_FALSE(Shl[1]);
}
```

Test class: llvm/utils/unittest/googletest/include/gtest/gtest.h

```
class GTEST_API_ Test {
public:
 // The d'tor is virtual as we intend to inherit from Test.
 virtual ~Test();
...
protected:
 // Creates a Test object.
 Test();

 // Sets up the test fixture.
 virtual void SetUp();

 // Tears down the test fixture.
 virtual void TearDown();
...
private:
 // Runs the test after the test fixture has been set up.
 //
 // A sub-class must implement this to define the test logic.
 //
 // DO NOT OVERRIDE THIS FUNCTION DIRECTLY IN A USER PROGRAM.
```

```
 // Instead, use the TEST or TEST_F macro.
 virtual void TestBody() = 0;

 // Sets up, executes, and tears down the test.
 void Run();
…
};
```

Test 를 생성하는 factory: unittest/googletest/include/gtest/internal/gtest-internal.h

```
// Defines the abstract factory interface that creates instances
// of a Test object.
class TestFactoryBase {
public:
 virtual ~TestFactoryBase() {}

 // Creates a test instance to run. The instance is both created and destroyed
 // within TestInfoImpl::Run()
 virtual Test* CreateTest() = 0;

protected:
 TestFactoryBase() {}

private:
 GTEST_DISALLOW_COPY_AND_ASSIGN_(TestFactoryBase);
};
```

TestFactoryBase is Inherited by:

```
// This class provides implementation of TeastFactoryBase interface.
// It is used in TEST and TEST_F macros.
template <class TestClass>
class TestFactoryImpl : public TestFactoryBase {
public:
 virtual Test* CreateTest() { return new TestClass; }
};

// Stores a parameter value and later creates tests parameterized with that
// value.
template <class TestClass>
class ParameterizedTestFactory : public TestFactoryBase {
public:
 typedef typename TestClass::ParamType ParamType;
```

```
 explicit ParameterizedTestFactory(ParamType parameter) :
     parameter_(parameter) {}
 virtual Test* CreateTest() {
   TestClass::SetParam(&parameter_);
   return new TestClass();
 }

private:
 const ParamType parameter_;

 GTEST_DISALLOW_COPY_AND_ASSIGN_(ParameterizedTestFactory);
};
```

TestFactoryBase is used by: TestInfo.
- TestInfo는 특정 Test 하나를 생성 -> 실행 -> 파괴합니다
- TestInfo가 여러 개 모인 것이 class TestCase 입니다
- TestCase가 여러 개 모인 것이 class UnitTest 입니다
- UnitTest는 singleton class입니다

googletest/include/gtest/gtest.h

```
class GTEST_API_ TestInfo {

...
 internal::TestFactoryBase* const factory_;  // The factory that creates
                                             // the test object

...
}
```

googletest/src/gtest.cc

```
// Creates the test object, runs it, records its result, and then
// deletes it.
void TestInfo::Run() {
 if (!should_run_) return;

 // Tells UnitTest where to store test result.
 internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
 impl->set_current_test_info(this);

 TestEventListener* repeater = UnitTest::GetInstance()->listeners().repeater();

 // Notifies the unit test event listeners that a test is about to start.
```

```
  repeater->OnTestStart(*this);

  const TimeInMillis start = internal::GetTimeInMillis();

  impl->os_stack_trace_getter()->UponLeavingGTest();

  // Creates the test object.
  Test* const test = internal::HandleExceptionsInMethodIfSupported(
      factory_, &internal::TestFactoryBase::CreateTest,
      "the test fixture's constructor");

  // Runs the test only if the test object was created and its
  // constructor didn't generate a fatal failure.
  if ((test != NULL) && !Test::HasFatalFailure()) {
    // This doesn't throw as all user code that can throw are wrapped into
    // exception handling code.
    test->Run();
  }
  // Deletes the test object.
  impl->os_stack_trace_getter()->UponLeavingGTest();
  internal::HandleExceptionsInMethodIfSupported(
      test, &Test::DeleteSelf_, "the test fixture's destructor");

  result_.set_elapsed_time(internal::GetTimeInMillis() - start);

  // Notifies the unit test event listener that a test has just finished.
  repeater->OnTestEnd(*this);

  // Tells UnitTest to stop associating assertion results to this
  // test.
  impl->set_current_test_info(NULL);
}
```

2. Builder

- IR Instruction 객체를 생성하는 클래스 (IRBuilder)에서 쓰입니다.

- Instruction 생성시
    (1) constant folding이 되는지 체크해서 가능하다면 fold된 상수를 반환
    (2) 아니면 새 instruction을 basic block에 삽입해줍니다.

- IRBuilder와 IRBuilderBase가 있음
    (1) IRBuilder는 FolderTy와 InserterTy 타입을 템플릿으로 강제함으로써, 올바른 IRBuilderBase객체가 만들어지고 있음을 컴파일 타임에 체크할 수 있습니다
    (2) InserterTy가 parameteric하게 주어지기 때문에, constructor를 호출할 수 있습니다
    (3) IRBuilder를 실수로 복사하는 것을 막아줍니다

llvm/include/llvm/IR/IRBuilder.h

```
/// Common base class shared among various IRBuilders.
class IRBuilderBase {
…
public:
 IRBuilderBase(LLVMContext &context, const IRBuilderFolder &Folder,
               const IRBuilderDefaultInserter &Inserter,
               MDNode *FPMathTag, ArrayRef<OperandBundleDef> OpBundles)
     : Context(context), Folder(Folder), Inserter(Inserter),
       DefaultFPMathTag(FPMathTag), IsFPConstrained(false),
       DefaultConstrainedExcept(fp::ebStrict),
       DefaultConstrainedRounding(fp::rmDynamic),
       DefaultOperandBundles(OpBundles) {
   ClearInsertionPoint();
 }
…
public:
 /// Insert and return the specified instruction.
 template<typename InstTy>
 InstTy *Insert(InstTy *I, const Twine &Name = "") const {
   Inserter.InsertHelper(I, Name, BB, InsertPt);
   SetInstDebugLocation(I);
   return I;
 }

 Value *CreateAdd(Value *LHS, Value *RHS, const Twine &Name = "",
                  bool HasNUW = false, bool HasNSW = false) {
   if (auto *LC = dyn_cast<Constant>(LHS))
     if (auto *RC = dyn_cast<Constant>(RHS))
       return Insert(Folder.CreateAdd(LC, RC, HasNUW, HasNSW), Name);
```

```
   return CreateInsertNUWNSWBinOp(Instruction::Add, LHS, RHS, Name,
                                   HasNUW, HasNSW);
 }
...
}

template <typename FolderTy = ConstantFolder,
          typename InserterTy = IRBuilderDefaultInserter>
class IRBuilder : public IRBuilderBase {
private:
 FolderTy Folder;
 InserterTy Inserter;

public:
 IRBuilder(LLVMContext &C, FolderTy Folder, InserterTy Inserter = InserterTy(),
           MDNode *FPMathTag = nullptr,
           ArrayRef<OperandBundleDef> OpBundles = None)
     : IRBuilderBase(C, this->Folder, this->Inserter, FPMathTag, OpBundles),
       Folder(Folder), Inserter(Inserter) {}
...

 /// Avoid copying the full IRBuilder. Prefer using InsertPointGuard
 /// or FastMathFlagGuard instead.
 IRBuilder(const IRBuilder &) = delete;
...
}
```

사용 예:
llvm/lib/Transforms/InstCombine/InstCombineInternal.h

```
class LLVM_LIBRARY_VISIBILITY InstCombiner
   : public InstVisitor<InstCombiner, Instruction *> {
 // FIXME: These members shouldn't be public.
public:
 /// A worklist of the instructions that need to be simplified.
 InstCombineWorklist &Worklist;

 /// An IRBuilder that automatically inserts new instructions into the
 /// worklist.
 using BuilderTy = IRBuilder<TargetFolder, IRBuilderCallbackInserter>;
 BuilderTy &Builder;
...
```

llvm/lib/Transforms/InstCombine/InstructionCombining.cpp

```
bool InstCombiner::run() {

...
   // Now that we have an instruction, try combining it to simplify it.
   Builder.SetInsertPoint(I);

...
   if (Instruction *Result = visit(*I)) {

...
}

Instruction *InstCombiner::visitGetElementPtrInst(GetElementPtrInst &GEP) {

...
     *I = Builder.CreateIntCast(*I, NewIndexType, true);

...
}
```

3. Factory Method

- 존재하지 않음

4. Prototype

- 맨 처음 Prototype을 만드는 부분: 존재하지 않음

- clone을 하는 부분: LLVM IR의 Instruction

llvm/include/llvm/IR/Instruction.h

```
class Instruction : public User,
                    public ilist_node_with_parent<Instruction, BasicBlock> {
...
 /// Create a copy of 'this' instruction that is identical in all ways except
 /// the following:
 ///   * The instruction has no parent
 ///   * The instruction has no name
 ///
 Instruction *clone() const;

private:
 /// Create a copy of this instruction.
 Instruction *cloneImpl() const;
}
```

Instruction.cpp

```
Instruction *Instruction::clone() const {
 Instruction *New = nullptr;
 switch (getOpcode()) {
 default:
   llvm_unreachable("Unhandled Opcode.");
#define HANDLE_INST(num, opc, clas)                                          \
 case Instruction::opc:                                                      \
   New = cast<clas>(this)->cloneImpl();                                      \
   break;
#include "llvm/IR/Instruction.def"
#undef HANDLE_INST
 }
```

```
 New->SubclassOptionalData = SubclassOptionalData; // e.g. nsw
 New->copyMetadata(*this);
 return New;
}


Instruction *Instruction::cloneImpl() const {
 llvm_unreachable("Subclass of Instruction failed to implement cloneImpl");
}
```

Instructions.cpp

```
…

FreezeInst *FreezeInst::cloneImpl() const {
 return new FreezeInst(getOperand(0));
}
```

- clone 함수의 문제점: clone()의 타입이 자기자신이 아니다.
- C++ 한정 해결책: CRTP pattern

```
class Parent<T> {
 virtual T* clone() = 0;
}
class Child : Parent<Child> {
  Child* clone() { return new Child(); }
}
```

5. Singleton Pattern

- LLVMContext 객체는 thread마다 하나가 존재
- Singleton으로 정의한 이유: Type, Constant 등의 객체가 단 한번씩만 생성되도록 관리하고 싶기 때문입니다.

llvm/include/llvm/IR/LLVMContext.h

```
/// This is an important class for using LLVM in a threaded context.  It
/// (opaquely) owns and manages the core "global" data of LLVM's core
/// infrastructure, including the type and constant uniquing tables.
/// LLVMContext itself provides no locking guarantees, so you should be careful
/// to have one context per thread.
class LLVMContext {
 LLVMContextImpl *const pImpl;
 LLVMContext();
 LLVMContext(LLVMContext &) = delete; (복사 불가)
 LLVMContext &operator=(const LLVMContext &) = delete; (다른 객체 대입 불가)
 ~LLVMContext();



...
 /// addModule - Register a module as being instantiated in this context.  If
 /// the context is deleted, the module will be deleted as well.
 void addModule(Module*);

 /// removeModule - Unregister a module from this context.
 void removeModule(Module*);
};
```

Type이 한 번만 생성되는 원리:

```
class IntegerType : public Type {
 friend class LLVMContextImpl;


protected:
 explicit IntegerType(LLVMContext &C, unsigned NumBits) : Type(C, IntegerTyID){
   setSubclassData(NumBits);
 }


public:
 static IntegerType *get(LLVMContext &C, unsigned NumBits);
…
}
```

llvm/lib/IR/Type.cpp

```
IntegerType *IntegerType::get(LLVMContext &C, unsigned NumBits) {
 assert(NumBits >= MIN_INT_BITS && "bitwidth too small");
 assert(NumBits <= MAX_INT_BITS && "bitwidth too large");

 // Check for the built-in integer types
 switch (NumBits) {
 case   1: return cast<IntegerType>(Type::getInt1Ty(C));
 case   8: return cast<IntegerType>(Type::getInt8Ty(C));
 case  16: return cast<IntegerType>(Type::getInt16Ty(C));
 case  32: return cast<IntegerType>(Type::getInt32Ty(C));
 case  64: return cast<IntegerType>(Type::getInt64Ty(C));
 case 128: return cast<IntegerType>(Type::getInt128Ty(C));
 default:
   break;
 }

 IntegerType *&Entry = C.pImpl->IntegerTypes[NumBits];

 if (!Entry)
   Entry = new (C.pImpl->Alloc) IntegerType(C, NumBits);

 return Entry;
}
```