

Unit Testing

Deep dive into mocking

What is Unit Testing?

Unit testing involves the testing of each **unit** or an **individual component** of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

A unit is a single testable part of a software system and tested during the **development** phase of the application software.

The purpose of unit testing is to test the correctness of **isolated** code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the **developers**.

What is Test Doubles?

According to the Google Testing Blog: Testing on the Toilet: Know your test doubles, test doubles are defined as follows:

A test double is an object that can stand in for a **real object** in a test, similar to how a stunt double stands in for an actor in a movie

```
public void testableMethod (A a, B b) {  
    // does something that needs to be tested  
}
```

Doubles that
don't simulate
behavior or
observe
interactions



Doubles that
simulate
behavior



Doubles that
observe
interactions



Dummy

Stub

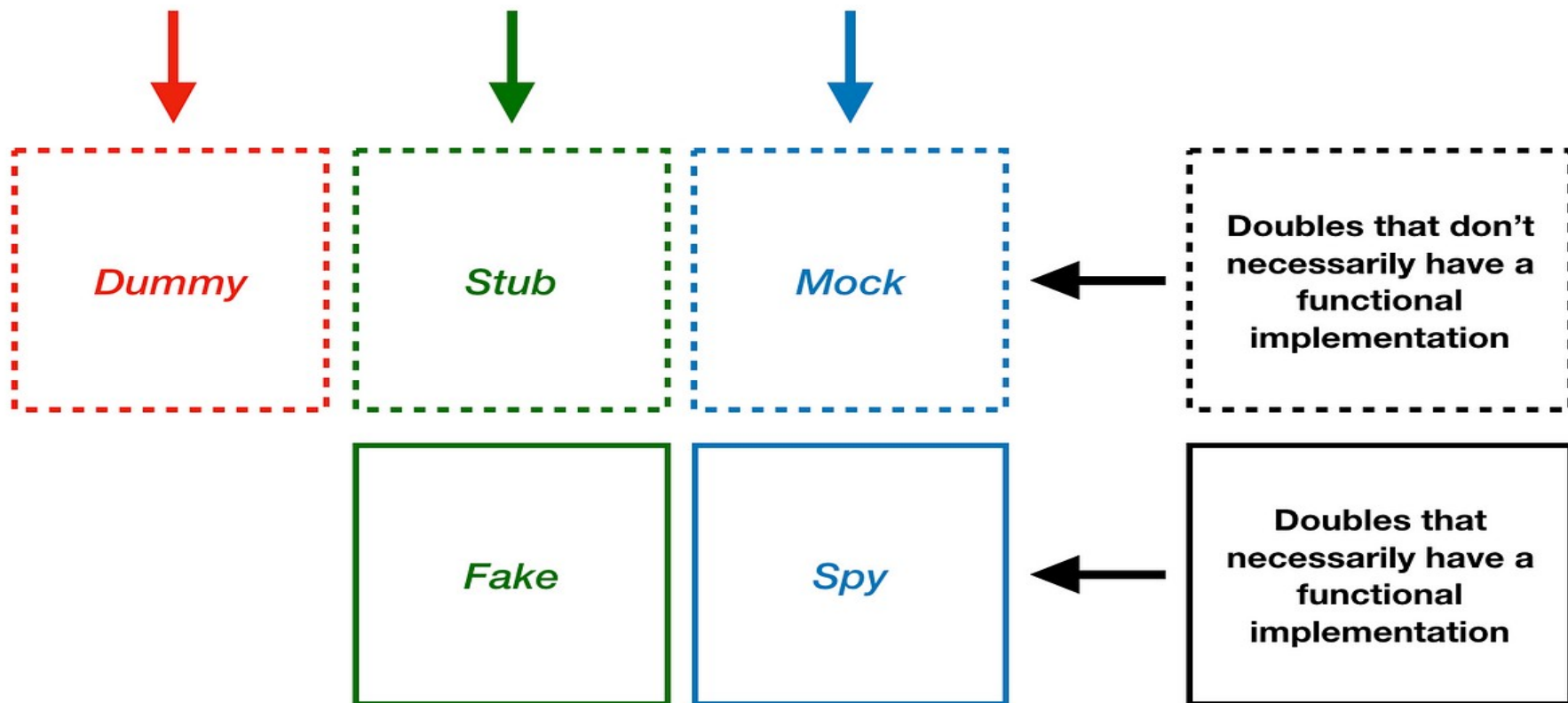
Mock

Doubles that don't
necessarily have a
functional
implementation

Fake

Spy

Doubles that
necessarily have a
functional
implementation



Stub	Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.
Mock	Mocks are objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed.
Fake	Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).
Dummy	Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.
Spy	Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.

Example



```
class GetItemListUseCase(  
    private val repository: ItemRepository,  
) {  
    operator fun invoke(): Result<List<Item>> {  
        return repository.getItemList()  
    }  
}
```



```
interface ItemRepository {  
    fun getItemList(): Result<List<Item>>  
    fun getItem(itemId: String): Result<Item>  
}  
  
open class ItemRepositoryImpl(  
    private val service: ItemService,  
) : ItemRepository {  
    override fun getItemList(): Result<List<Item>> {  
        return service.getItemList()  
    }  
  
    override fun getItem(itemId: String): Result<Item> {  
        if (itemId.isBlank()) {  
            return Result.failure(IllegalArgumentException())  
        }  
        return service.getItem(itemId)  
    }  
}
```


Stubs



```
class StubItemRepository(  
    private val itemList: List<Item> = emptyList(),  
    private val item: Item = Item(),  
) : ItemRepository {  
    override fun getItemList(): Result<List<Item>> {  
        return Result.success(itemList)  
    }  
  
    override fun getItem(itemId: String): Result<Item> {  
        return Result.success(item)  
    }  
}
```



```
class GetItemListUseCaseTest {  
    lateinit var useCase: GetItemListUseCase  
  
    @Test  
    fun `invoke return itemList`() {  
        val itemList = listOf(  
            Item("item1"),  
            Item("item2")  
        )  
        val repository = StubItemRepository(itemList)  
        useCase = GetItemListUseCase(repository)  
  
        val result = useCase.invoke()  
        Truth.assertThat(result.isSuccess).isTrue()  
        Truth.assertThat(result.getOrNull()).isEqualTo(itemList)  
    }  
}
```

Stub With Mockk

Stubs return a canned value.
Therefore, other Test Doubles also have a
part of a Stub feature.

```
class GetItemListUseCaseTest {
    lateinit var useCase: GetItemListUseCase


    @Test
    fun `invoke return itemList`() {
        val itemList = listOf(
            Item("item1"),
            Item("item2")
        )
        val repository = mockk<ItemRepository> {
            every { getItemList() } returns Result.success(itemList)
        }
        useCase = GetItemListUseCase(repository)

        val result = useCase.invoke()
        Truth.assertThat(result.isSuccess).isTrue()
        Truth.assertThat(result.getOrNull()).isEqualTo(itemList)
    }
}
```

Mock

Mocks are similar to Stubs, but Mocks can check if the specified method is called or not.

Also, Mocks can throw an exception if an unexpected function is called.



```
class MockItemRepository(  
    private val itemList: List<Item> = emptyList(),  
    ) : ItemRepository {  
    var isGetItemListCalled: Boolean = false  
  
    override fun getItemList(): Result<List<Item>> {  
        isGetItemListCalled = true  
        return Result.success(itemList)  
    }  
  
    override fun getItem(itemId: String): Result<Item> {  
        error("This function is not mocked")  
    }  
}
```



```
class GetItemListUseCaseTest {  
    lateinit var useCase: GetItemListUseCase  
  
    @Test  
    fun `invoke return itemList`() {  
        val itemList = listOf(  
            Item("item1"),  
            Item("item2")  
        )  
        val repository = MockItemRepository(itemList)  
        useCase = GetItemListUseCase(repository)  
  
        val result = useCase.invoke()  
  
        Truth.assertThat(repository.isGetItemListCalled).isTrue()  
    }  
}
```


Mock with Mockk

```
class GetItemListUseCaseTest {
    lateinit var useCase: GetItemListUseCase

    @Test
    fun `invoke return itemList`() {
        val itemList = listOf(
            Item("item1"),
            Item("item2")
        )
        val repository = mockk<ItemRepository> {
            every { getItemList() } returns Result.success(itemList)
        }
        useCase = GetItemListUseCase(repository)

        val result = useCase.invoke()

        verify(exactly = 1) { repository.getItemList() }
        confirmVerified(repository)
    }
}
```

Spy

Spies are very similar to Mocks, but Spies don't throw an exception if an unexpected function is called.

Therefore, Spies are often used as wrappers for real objects.



```
class SpyItemRepository(
    service: ItemService,
    private val itemList: List<Item> = emptyList(),
) : ItemRepositoryImpl(service) {
    var isGetItemListCalled: Boolean = false

    override fun getItemList(): Result<List<Item>> {
        isGetItemListCalled = true
        // You can return super.getItemList() as it is if you don't need the Stub feature.
        return Result.success(itemList)
    }
}
```

```
class GetItemListUseCaseTest {
    lateinit var useCase: GetItemListUseCase

    @Test
    fun `invoke return itemList`() {
        val itemList = listOf(
            Item("item1"),
            Item("item2")
        )
        val repository = SpyItemRepository(DummyItemService(), itemList)
        useCase = GetItemListUseCase(repository)

        val result = useCase.invoke()

        Truth.assertThat(repository.isGetItemListCalled).isTrue()
    }
}
```

Spy with Mockk



```
class GetItemListUseCaseTest {  
    lateinit var useCase: GetItemListUseCase  
  
    @Test  
    fun `invoke return itemList`() {  
        val itemList = listOf(  
            Item("item1"),  
            Item("item2")  
        )  
        val repository = spyk(ItemRepositoryImpl(DummyItemService())) {  
            every { getItemList() } returns Result.success(itemList)  
        }  
        useCase = GetItemListUseCase(repository)  
  
        val result = useCase.invoke()  
  
        verify(exactly = 1) { repository.getItemList() }  
        confirmVerified(repository)  
    }  
}
```

Spy vs mock

Both can be used to mock methods or fields. The difference is that in mock, you are creating a **complete mock** or **fake object** while in spy, there is the **real object** and you just spying or stubbing specific methods of it.

When using mock objects, the default behavior of the method when not stub is do nothing. Simple means, if its a void method, then it will do nothing when you call the method or if its a method with a return then it may return null, empty or the default value.

While in spy objects, of course, since it is a real method, when you are not stubbing the method, then it will call the real method behavior. If you want to change and mock the method, then you need to stub it.

Fake

Fakes are similar to Stubs, but Fakes have working implementations that are the same as real objects.

In this case, a Fake implementation for `getItemList()` is the same as the Stub.

But `getItem(itemId: String)` of the real object has a validation for `itemId`, so the Fake should have that as well.

Also, `getItem(itemId: String)` should return a specified item by `itemId`, so you need to implement that as well, maybe the server side has a similar logic

```
class FakeItemRepository : ItemRepository {
    private val item1 = Item("item1")
    private val item2 = Item("item2")
    private val itemList: List<Item> = listOf(item1,item2)

    override fun getItemList(): Result<List<Item>> {
        return Result.success(itemList)
    }

    override fun getItem(itemId: String): Result<Item> {
        if (itemId.isBlank()) { return Result.failure(IllegalArgumentException()) }
        // service or server side has a similar logic
        val item = itemList.firstOrNull { it.id == itemId }
        return if (item == null) {
            Result.failure(IllegalArgumentException())
        } else {
            Result.success(item)
        }
    }
}
```

Dummy

Dummies are used just to avoid compile errors. This is not important.



```
class DummyItemRepository : ItemRepository {  
    override fun getItemList(): Result<List<Item>> { TODO("Not yet implemented") }  
  
    override fun getItem(itemId: String): Result<Item> { TODO("Not yet implemented") }  
}  
  
class GetItemListUseCaseTest {  
    lateinit var useCase: GetItemListUseCase  
  
    @Test  
    fun `test something`() {  
        val repository = DummyRepository()  
        useCase = GetItemListUseCase(repository)  
        ...  
    }  
}
```

Dummy with Mockk



```
class GetItemListUseCaseTest {  
    lateinit var useCase: GetItemListUseCase  
  
    @Test  
    fun `test something`() {  
        val repository = mockk<ItemRepository>()  
        useCase = GetItemListUseCase(repository)  
        ...  
    }  
}
```

Which is the best Test Doubles to use?

In my opinion, it depends because each Test Double has pros and cons.

Stubs are useful for all tests, but sometimes they are not sufficient for testing.

Mocks and Spies are especially useful for Unit Tests, because we often want to verify the method calling.

Fakes are useful for Integration Tests and UI Tests using Robolectric or Espresso, because we just want to check the behavior of the application.

Therefore, I don't think you need to limit yourself to using only one type.

You might need some rules or policies, but Test Doubles are just a tool, and not the purpose.

Deep Dive Into





THANK YOU

Credit by : mohsen abedini, 2023.