Put Your Tests on a Diet

Testing the Behavior and Not the Implementation

Stelios Frantzeskakis - Perry Street Software

droidcon London 2023

Do we still believe in tests?

We’ve been doing it wrong

Hi, I’m Stelios Frantzeskakis

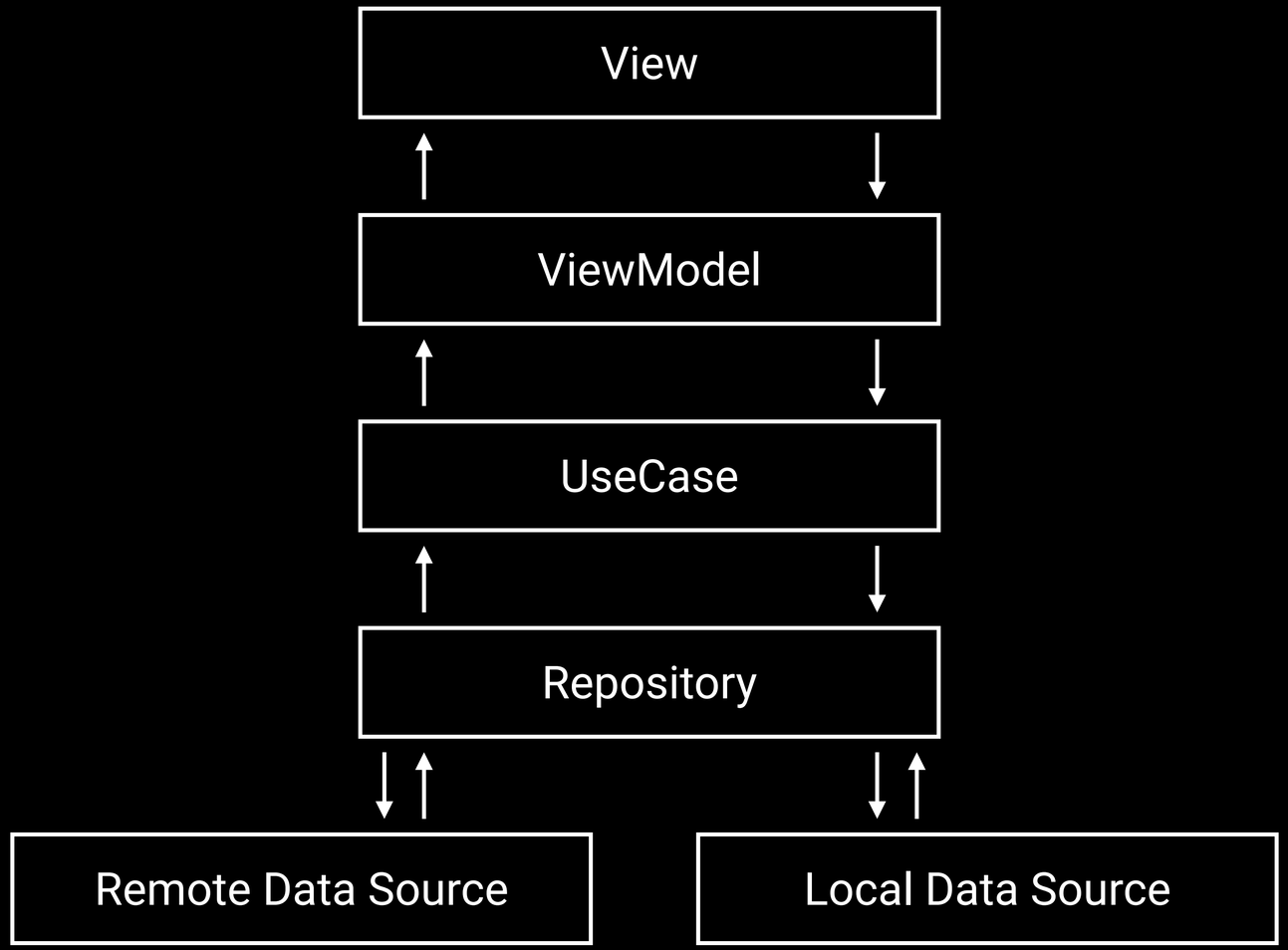
Staff Engineer at Perry Street Software, publisher of SCRUFF & Jack’d, serving more than 30M members

Working with Android since the release of Android Gingerbread

Passionate about Testing & Architecture

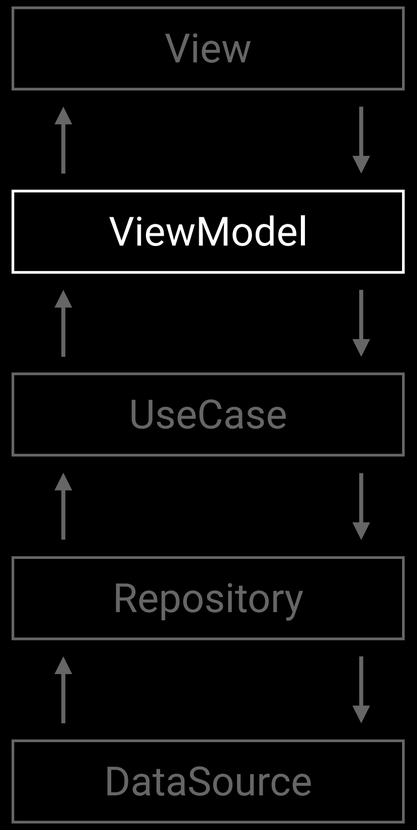
@SteliosFran

Modern app architecture

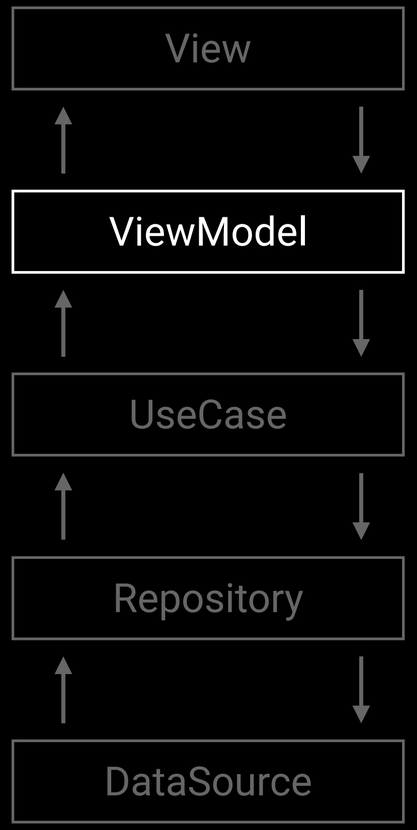


Sample chat application

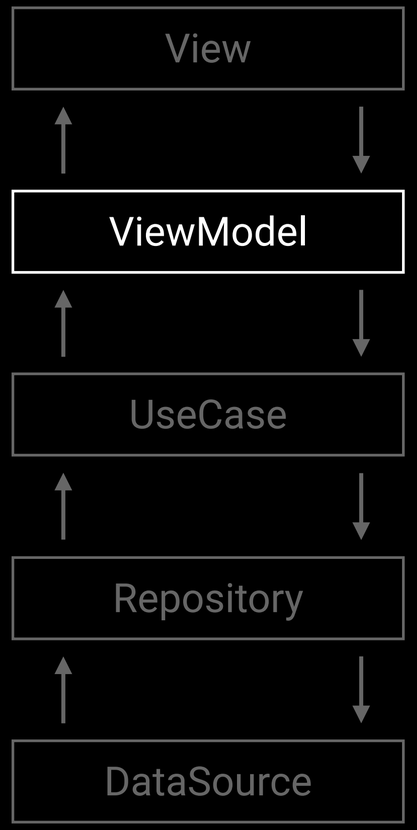
```
class ChatViewModel(
    private val getChatMessagesUseCase: GetChatMessagesUseCase,
    private val sendTextMessageUseCase: SendTextMessageUseCase,
    private val user: User,
    private val coroutineDispatcherProvider: CoroutineDispatcherProvider
) : ViewModel() {

    data class State(
        val messages: List<MessageUiModel>
    )

    private val _state = MutableStateFlow(State(emptyList()))
    val state: StateFlow<State> = _state.asStateFlow()

    init {
        viewModelScope.launch(coroutineDispatcherProvider.main) {
            getChatMessagesUseCase(user).map {
                State(MessageUiModelMapper.fromMessagesList(it))
            }.collect { newState ->
                _state.value = newState
            }
        }
    }

    fun onTextMessageSent(text: String) = viewModelScope.launch(
        coroutineDispatcherProvider.main
    ) {
        sendTextMessageUseCase(text, user)
    }
}
```



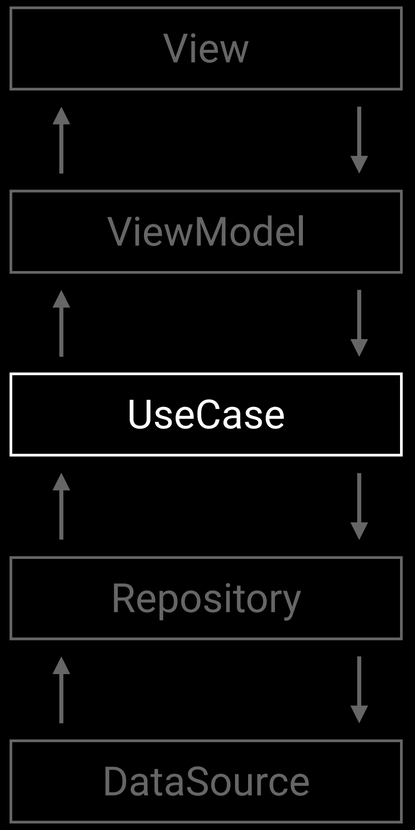
```kotlin
class ChatViewModel(
    private val getChatMessagesUseCase: GetChatMessagesUseCase,
    private val sendTextMessageUseCase: SendTextMessageUseCase,
    private val user: User,
    private val coroutineDispatcherProvider: CoroutineDispatcherProvider
) : ViewModel() {

    data class State(
        val messages: List<MessageUiModel>
    )

    private val _state = MutableStateFlow(State(emptyList()))
    val state: StateFlow<State> = _state.asStateFlow()

    init {
        viewModelScope.launch(coroutineDispatcherProvider.main) {
            getChatMessagesUseCase(user).map {
                State(MessageUiModelMapper.fromMessagesList(it))
            }.collect { newState ->
                _state.value = newState
            }
        }
    }

    fun onTextMessageSent(text: String) = viewModelScope.launch(
        coroutineDispatcherProvider.main
    ) {
        sendTextMessageUseCase(text, user)
    }
}
```



```
class ChatViewModel(
    private val getChatMessagesUseCase: GetChatMessagesUseCase,
    private val sendTextMessageUseCase: SendTextMessageUseCase,
    private val user: User,
    private val coroutineDispatcherProvider: CoroutineDispatcherProvider
) : ViewModel() {

    data class State(
        val messages: List<MessageUiModel>
    )

    private val _state = MutableStateFlow(State(emptyList()))
    val state: StateFlow<State> = _state.asStateFlow()

    init {
        viewModelScope.launch(coroutineDispatcherProvider.main) {
            getChatMessagesUseCase(user).map {
                State(MessageUiModelMapper.fromMessagesList(it))
            }.collect { newState ->
                _state.value = newState
            }
        }
    }

    fun onTextMessageSent(text: String) = viewModelScope.launch(
        coroutineDispatcherProvider.main
    ) {
        sendTextMessageUseCase(text, user)
    }
}
```

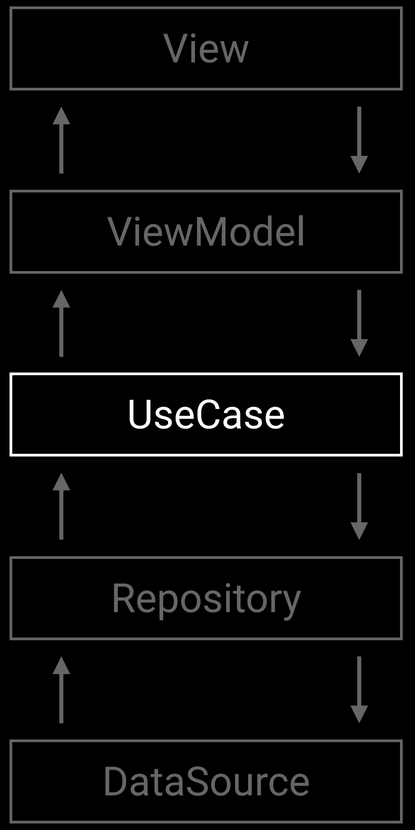


```kotlin
class GetChatMessagesUseCase(
    private val chatMessagesRepository: ChatMessagesRepository
) {

    operator fun invoke(user: User): Flow<List<Message>> {
        return chatMessagesRepository.getChatMessages(user).map { messagesList ->
            messagesList.sortedBy { it.date.time }
        }
    }
}


class SendTextMessageUseCase(
    private val chatMessagesRepository: ChatMessagesRepository
) {

    suspend operator fun invoke(text: String, user: User) {
        if (isValid(text)) {
            val message = MessageFactory.fromText(text)
            chatMessagesRepository.sendChatMessage(message, user)
        } else {
            throw InvalidMessageException("Message is too long")
        }
    }

    private fun isValid(text: String): Boolean {
        return text.length < 180
    }
}
```

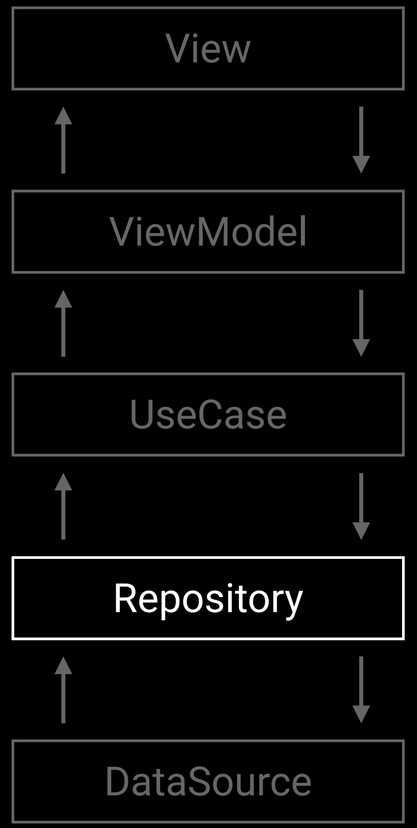


```kotlin
class GetChatMessagesUseCase(
    private val chatMessagesRepository: ChatMessagesRepository
) {

    operator fun invoke(user: User): Flow<List<Message>> {
        return chatMessagesRepository.getChatMessages(user).map { messagesList ->
            messagesList.sortedBy { it.date.time }
        }
    }
}
```

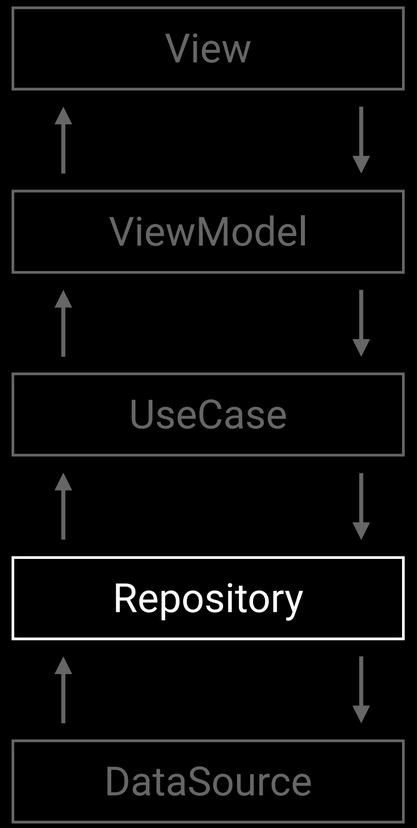
```kotlin
class SendTextMessageUseCase(
    private val chatMessagesRepository: ChatMessagesRepository
) {

    suspend operator fun invoke(text: String, user: User) {
        if (isValid(text)) {
            val message = MessageFactory.fromText(text)
            chatMessagesRepository.sendChatMessage(message, user)
        } else {
            throw InvalidMessageException("Message is too long")
        }
    }

    private fun isValid(text: String): Boolean {
        return text.length < 180
    }
}
```



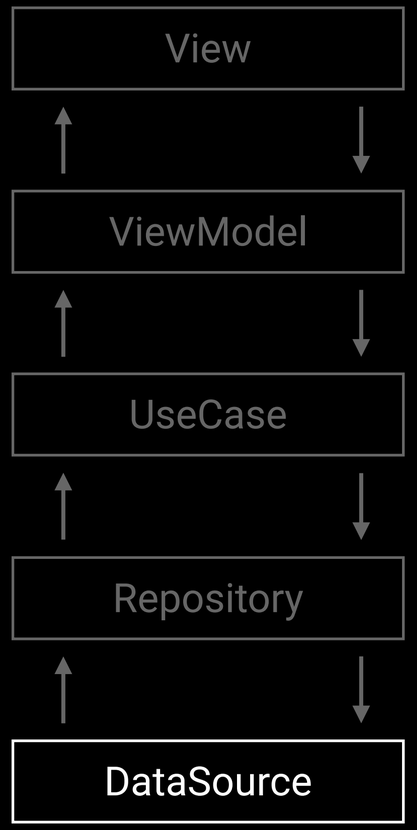
```kotlin
class ChatMessagesRepository(
    private val chatDataSource: ChatDataSource
) {
    private val _cachedChatMessages = MutableStateFlow<List<Message>>(emptyList())

    fun getChatMessages(user: User): Flow<List<Message>> = flow {
        if (_cachedChatMessages.value.isEmpty()) {
            _cachedChatMessages.emit(chatDataSource.getChatMessages(user))
        }
        emitAll(_cachedChatMessages)
    }

    suspend fun sendChatMessage(message: Message, user: User) {
        _cachedChatMessages.apply { value = value + message }
        chatDataSource.sendChatMessage(message, user)
    }
}
```



```kotlin
class ChatMessagesRepository(
    private val chatDataSource: ChatDataSource
) {
    private val _cachedChatMessages = MutableStateFlow<List<Message>>(emptyList())

    fun getChatMessages(user: User): Flow<List<Message>> = flow {
        if (_cachedChatMessages.value.isEmpty()) {
            _cachedChatMessages.emit(chatDataSource.getChatMessages(user))
        }
        emitAll(_cachedChatMessages)
    }

    suspend fun sendChatMessage(message: Message, user: User) {
        _cachedChatMessages.apply { value = value + message }
        chatDataSource.sendChatMessage(message, user)
    }
}
```



```
interface ChatDataSource {
    suspend fun getChatMessages(user: User): List<Message>
    suspend fun sendChatMessage(message: Message, user: User)
}
```



How do you write unit tests for this architecture?

Just mock every layer

```kotlin
class ChatViewModelTest {

    private val getChatMessagesUseCase: GetChatMessagesUseCase = mockk()
    private val sendTextMessageUseCase: SendTextMessageUseCase = mockk()
    private val user = User(id = "test_id", name = "test name")
    private val firstMessage = Message("id1", "1st message", Date().minusDays(1), Message.Type.Received)
    private val secondMessage = Message("id2", "2nd message", Date(), Message.Type.Received)
    private val messagesList = listOf(firstMessage, secondMessage)

    private val viewModel by lazy {
        ChatViewModel(
            getChatMessagesUseCase,
            sendTextMessageUseCase,
            user,
            TestCoroutineDispatcherProvider()
        )
    }

    // ..
}
```

```kotlin
class ChatViewModelTest {

    // ...

    @Test
    fun `it emits the state`() {
        every { getChatMessagesUseCase(user) } returns flowOf(messagesList)

        viewModel.state.value shouldBeEqual ChatViewModel.State(
            messages = listOf(
                MessageUiModel(
                    text = "1st message", type = Message.Type.Received
                ),
                MessageUiModel(
                    text = "2nd message", type = Message.Type.Received
                )
            )
        )
    }

    @Test
    fun `it sends a text message`() {
        viewModel.onTextMessageSent("text message")

        coVerify(exactly = 1) { sendTextMessageUseCase(text = "text message", user = user) }
    }
}
```

```kotlin
class ChatViewModelTest {

    // ...

    @Test
    fun `it emits the state`() {
        every { getChatMessagesUseCase(user) } returns flowOf(messagesList)

        viewModel.state.value shouldBeEqual ChatViewModel.State(
            messages = listOf(
                MessageUiModel(
                    text = "1st message", type = Message.Type.Received
                ),
                MessageUiModel(
                    text = "2nd message", type = Message.Type.Received
                )
            )
        )
    }

    @Test
    fun `it sends a text message`() {
        viewModel.onTextMessageSent("text message")

        coVerify(exactly = 1) { sendTextMessageUseCase(text = "text message", user = user) }
    }
}
```

```kotlin
class GetChatMessagesUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")
    private val firstMessage = Message("id1", "1st message", Date().minusDays(1), Message.Type.Received)
    private val secondMessage = Message("id2", "2nd message", Date(), Message.Type.Received)
    private val messagesList = listOf(secondMessage, firstMessage)

    private val getChatMessagesUseCase = GetChatMessagesUseCase(chatMessagesRepository)

    @Test
    fun `it emits the list of chat messages`() {
        every { chatMessagesRepository.getChatMessages(user) } returns flowOf(messagesList)

        runTest {
            getChatMessagesUseCase(user).first() shouldBeEqual listOf(
                firstMessage, secondMessage
            )
        }
    }
}
```

```kotlin
class GetChatMessagesUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")
    private val firstMessage = Message("id1", "1st message", Date().minusDays(1), Message.Type.Received)
    private val secondMessage = Message("id2", "2nd message", Date(), Message.Type.Received)
    private val messagesList = listOf(secondMessage, firstMessage)

    private val getChatMessagesUseCase = GetChatMessagesUseCase(chatMessagesRepository)

    @Test
    fun `it emits the list of chat messages`() {
        every { chatMessagesRepository.getChatMessages(user) } returns flowOf(messagesList)

        runTest {
            getChatMessagesUseCase(user).first() shouldBeEqual listOf(
                firstMessage, secondMessage
            )
        }
    }
}
```

```kotlin
class SendTextMessageUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")

    private val sendTextMessageUseCase = SendTextMessageUseCase(chatMessagesRepository)

    @Test
    fun `it sends a valid text message`() {
        val validTextMessage = "text message"
        val message = MessageFactory.fromText(validTextMessage)
        coEvery { chatMessagesRepository.sendChatMessage(message, user) } just Runs

        runTest {
            sendTextMessageUseCase(validTextMessage, user)
            coVerify(exactly = 1) { chatMessagesRepository.sendChatMessage(message, user) }
        }
    }

    @Test
    fun `it does not send an invalid text message`() {
        val invalidTextMessage = (1..180).joinToString("") { "a" }
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(invalidTextMessage, user)
            coVerify(exactly = 0) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }
}
```

```kotlin
class SendTextMessageUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")

    private val sendTextMessageUseCase = SendTextMessageUseCase(chatMessagesRepository)

    @Test
    fun `it sends a valid text message`() {
        val validTextMessage = "text message"
        val message = MessageFactory.fromText(validTextMessage)
        coEvery { chatMessagesRepository.sendChatMessage(message, user) } just Runs

        runTest {
            sendTextMessageUseCase(validTextMessage, user)
            coVerify(exactly = 1) { chatMessagesRepository.sendChatMessage(message, user) }
        }
    }

    @Test
    fun `it does not send an invalid text message`() {
        val invalidTextMessage = (1..180).joinToString("") { "a" }
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(invalidTextMessage, user)
            coVerify(exactly = 0) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }
}
```

```kotlin
class SendTextMessageUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")

    private val sendTextMessageUseCase = SendTextMessageUseCase(chatMessagesRepository)

    @Test
    fun `it sends a valid text message`() {
        val validTextMessage = "text message"
        val message = MessageFactory.fromText(validTextMessage)
        coEvery { chatMessagesRepository.sendChatMessage(message, user) } just Runs

        runTest {
            sendTextMessageUseCase(validTextMessage, user)
            coVerify(exactly = 1) { chatMessagesRepository.sendChatMessage(message, user) }
        }
    }

    @Test
    fun `it does not se
        val invalidText
        coEvery { chatM                                                    st Runs

        runTest {
            sendTextMes
            coVerify(ex                                                    (any(), user) }
        }
    }
}
```

```kotlin
object MessageFactory {
    fun fromText(text: String): Message {
        return Message(
            id = UUID.randomUUID().toString(),
            text = text,
            date = Date(),
            type = Message.Type.Pending
        )
    }
}
```

```kotlin
class SendTextMessageUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")

    private val sendTextMessageUseCase = SendTextMessageUseCase(chatMessagesRepository)

    @Test
    fun `it sends a valid text message`() {
        val validTextMessage = "text message"
        // val message = MessageFactory.fromText(validTextMessage)
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(validTextMessage, user)
            coVerify(exactly = 1) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }

    @Test
    fun `it does not send an invalid text message`() {
        val invalidTextMessage = (1..180).joinToString("") { "a" }
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(invalidTextMessage, user)
            coVerify(exactly = 0) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }
}
```

```kotlin
class SendTextMessageUseCaseTest {

    private val chatMessagesRepository: ChatMessagesRepository = mockk()
    private val user = User(id = "test_id", name = "test name")

    private val sendTextMessageUseCase = SendTextMessageUseCase(chatMessagesRepository)

    @Test
    fun `it sends a valid text message`() {
        val validTextMessage = "text message"
        // val message = MessageFactory.fromText(validTextMessage)
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(validTextMessage, user)
            coVerify(exactly = 1) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }

    @Test
    fun `it does not send an invalid text message`() {
        val invalidTextMessage = (1..180).joinToString("") { "a" }
        coEvery { chatMessagesRepository.sendChatMessage(any(), user) } just Runs

        runTest {
            sendTextMessageUseCase(invalidTextMessage, user)
            coVerify(exactly = 0) { chatMessagesRepository.sendChatMessage(any(), user) }
        }
    }
}
```

```
class ChatMessagesRepositoryTest {

    private val chatDataSource: ChatDataSource = mockk()
    private val user = User(id = "test_id", name = "test name")
    private val firstMessage = Message("id1", "1st message", Date().minusDays(1), Message.Type.Received)
    private val secondMessage = Message("id2", "2nd message", Date(), Message.Type.Received)
    private val messagesList = listOf(firstMessage, secondMessage)

    private val chatMessagesRepository = ChatMessagesRepository(chatDataSource)

      // ...
}
```

```kotlin
class ChatMessagesRepositoryTest {

    // ...

    @Test
    fun `it emits the list of chat messages`() {
        coEvery { chatDataSource.getChatMessages(user) } returns messagesList

        runTest {
            chatMessagesRepository.getChatMessages(user).first() shouldBeEqual messagesList
        }
    }

    @Test
    fun `it sends a chat message`() {
        val message = MessageFactory.fromText("text message")
        coEvery { chatDataSource.sendChatMessage(message, user) } just Runs

        runTest {
            chatMessagesRepository.sendChatMessage(message, user)
            coVerify(exactly = 1) { chatDataSource.sendChatMessage(message, user) }
        }
    }

    // ...
}
```

```kotlin
class ChatMessagesRepositoryTest {

    // ...

    @Test
    fun `it emits the list of chat messages`() {
        coEvery { chatDataSource.getChatMessages(user) } returns messagesList

        runTest {
            chatMessagesRepository.getChatMessages(user).first() shouldBeEqual messagesList
        }
    }

    @Test
    fun `it sends a chat message`() {
        val message = MessageFactory.fromText("text message")
        coEvery { chatDataSource.sendChatMessage(message, user) } just Runs

        runTest {
            chatMessagesRepository.sendChatMessage(message, user)
            coVerify(exactly = 1) { chatDataSource.sendChatMessage(message, user) }
        }
    }

    // ...
}
```

```
class ChatMessagesRepositoryTest {

    // ...

    @Test
    fun `it updates the list of chat messages after sending a message`() {
        val message = MessageFactory.fromText("text message")
        coEvery { chatDataSource.getChatMessages(user) } returns messagesList
        coEvery { chatDataSource.sendChatMessage(message, user) } just Runs

        runTest {
            chatMessagesRepository.getChatMessages(user).first() shouldBeEqual listOf(
                firstMessage, secondMessage
            )
            chatMessagesRepository.sendChatMessage(message, user)

            chatMessagesRepository.getChatMessages(user).first() shouldBeEqual listOf(
                firstMessage, secondMessage, message
            )
        }
    }
}
```

Problems of mocking every layer

Tests are tied to the implementation

Tests become overspecified

Mocks bind you to a specific dependency graph

Tests become the obstacle to refactoring

What do the experts say about mocks?

“Mockist tests are coupled to the implementation of a method. Coupling to the implementation interferes with refactoring, since implementation changes are much more likely to break tests.”

“I don’t see any compelling benefits for mockist TDD, and am concerned about the consequences of coupling tests to implementation.I really like the fact that while writing the test you focus on the result of the behavior, not how it’s done.”

Martin Fowler

“You have to mock out all the data pathways in the interaction; and that can be a complex task. This creates two problems. 1. The setup code can get extremely complicated. 2. The mocking structure become tightly coupled to implementation details causing many tests to break when those details are modified.”

“The need to mock every class interaction forces an explosion of polymorphic interfaces whose sole purpose is to allow mocking.”

“I recommend that you mock sparingly. Find a way to test — design a way to test — your code so that it doesn’t require a mock.”

Robert C. Martin (Uncle Bob)

“I mock almost nothing. If I can’t figure out how to test efficiently with the real stuff, I find another way of creating a feedback loop for myself.”

“If you have mocks returning mocks, returning mocks, your test is completely coupled to the implementation, not the interface, but the exact implementation. Of course you can’t change anything without breaking the test.”

Kent Beck

The BDD approach

Given - When - Then

```
Given("I have exchanged messages with a user") {
    fakeDataSourceResponse()

    Then("They are displayed") {
        viewModel.state.value shouldBeEqual ChatViewModel.State(
            messages = listOf(
                MessageUiModel(
                    text = "1st message", type = Message.Type.Received
                )
            )
        )
    }

    // ...
}
```

```
Given("I have exchanged messages with a user") {
    fakeDataSourceResponse()

    Then("They are displayed") {
        viewModel.state.value shouldBeEqual ChatViewModel.State(
            messages = listOf(
                MessageUiModel(
                    text = "1st message", type = Message.Type.Received
                )
            )
        )
    }

    // ...
}
```

```
Given("I have exchanged messages with a user") {
    // ...

    When("I send a text message") {
        var textMessage = ""
        beforeEach {
            viewModel.onTextMessageSent(textMessage)
        }

        And("It is valid") {
            textMessage = "2nd message"

            Then("It is displayed") {
                viewModel.state.value shouldBeEqual ChatViewModel.State(
                    messages = listOf(
                        MessageUiModel(
                            text = "1st message", type = Message.Type.Received
                        ),
                        MessageUiModel(
                            text = "2nd message", type = Message.Type.Pending
                        )
                    )
                )
            }
        }

        // ...
    }
}
```

```
Given("I have exchanged messages with a user") {
    // ...

    When("I send a text message") {
        var textMessage = ""
        beforeEach {
            viewModel.onTextMessageSent(textMessage)
        }

        And("It is valid") {
            textMessage = "2nd message"

            Then("It is displayed") {
                viewModel.state.value shouldBeEqual ChatViewModel.State(
                    messages = listOf(
                        MessageUiModel(
                            text = "1st message", type = Message.Type.Received
                        ),
                        MessageUiModel(
                            text = "2nd message", type = Message.Type.Pending
                        )
                    )
                )
            }
        }

        // ...
    }
}
```

```
Given("I have exchanged messages with a user") {
    // ...

    When("I send a text message") {
        var textMessage = ""
        beforeEach {
            viewModel.onTextMessageSent(textMessage)
        }

        And("It is valid") {
            textMessage = "2nd message"

            Then("It is displayed") {
                viewModel.state.value shouldBeEqual ChatViewModel.State(
                    messages = listOf(
                        MessageUiModel(
                            text = "1st message", type = Message.Type.Received
                        ),
                        MessageUiModel(
                            text = "2nd message", type = Message.Type.Pending
                        )
                    )
                )
            }
        }

        // ...
    }
}
```

```
Given("I have exchanged messages with a user") {
    // ...

    When("I send a text message") {
        var textMessage = ""
        beforeEach {
            viewModel.onTextMessageSent(textMessage)
        }

        // ...

        And("It is not valid") {
            textMessage = (1..180).joinToString("") { "a" }

            Then("It is not displayed") {
                viewModel.state.value shouldBeEqual ChatViewModel.State(
                    messages = listOf(
                        MessageUiModel(
                            "1st message", Message.Type.Received
                        )
                    )
                )
            }
        }
    }
}
```

```
Given("I have exchanged messages with a user") {
    // ...

    When("I send a text message") {
        var textMessage = ""
        beforeEach {
            viewModel.onTextMessageSent(textMessage)
        }

        // ...

        And("It is not valid") {
            textMessage = (1..180).joinToString("") { "a" }

            Then("It is not displayed") {
                viewModel.state.value shouldBeEqual ChatViewModel.State(
                    messages = listOf(
                        MessageUiModel(
                            "1st message", Message.Type.Received
                        )
                    )
                )
            }
        }
    }
}
```

Which layers did we test?

Is this a unit test or an integration test?

A unit test is a test that runs in isolation from other tests

Kent Beck

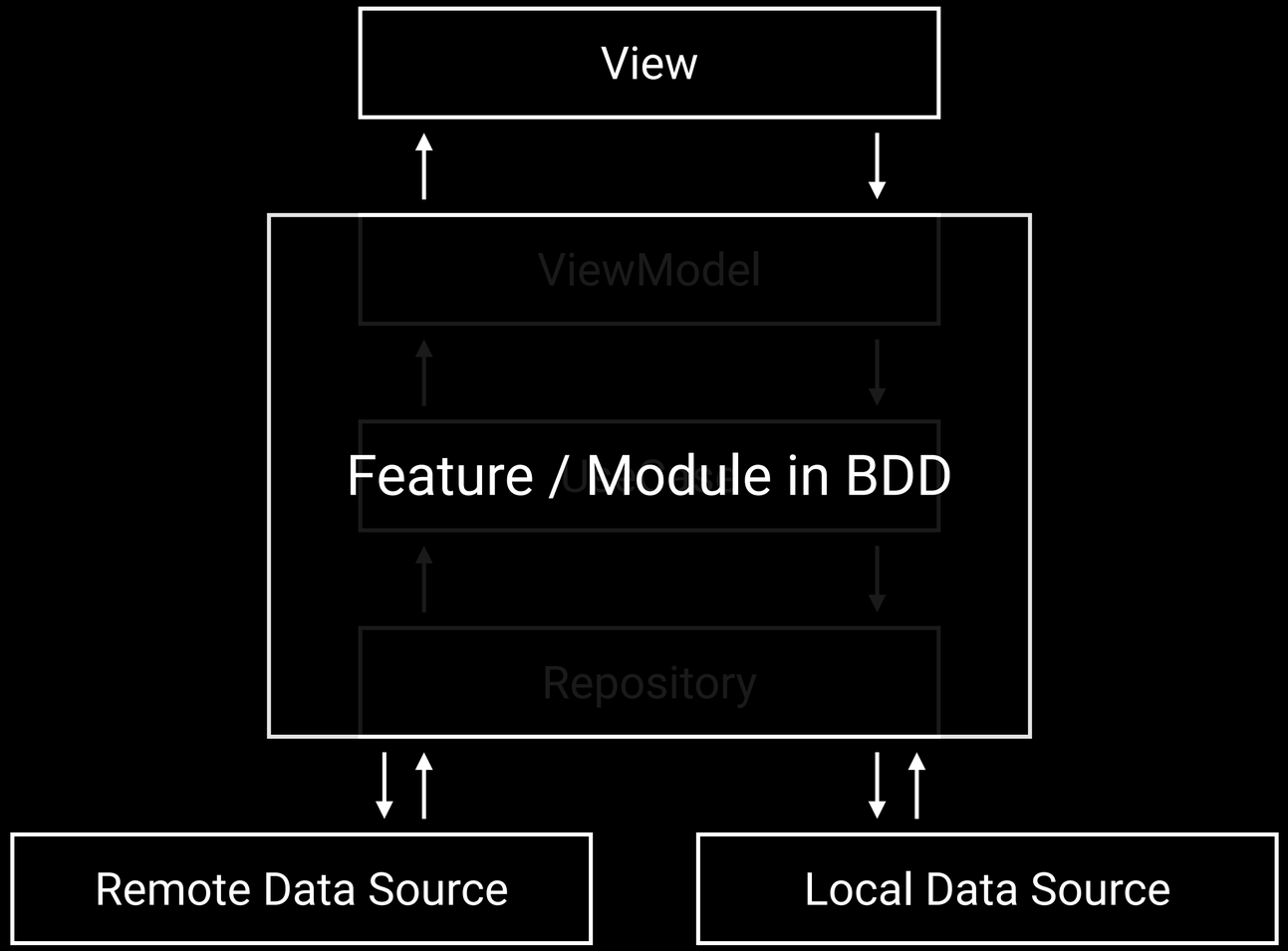
In sociable unit tests, the tested unit
can rely on other classes

Martin Fowler

The unit is an isolated module. It’s a black box where you talk to what is exposed.

Ian Cooper

Unit tests are not about testing a single class



When to mock?

Benefits of BDD

Refactoring won’t break the tests

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”

Martin Fowler

Tests are the documentation

TDD: Red - Green - Refactor

Less code to write and maintain

Additional resources

Classical and Mockist Testing - Martin Fowler

https://martinfowler.com/articles/mocksArentStubs.html#ClassicalAndMockistTesting

On the Diverse And Fantastical Shapes of Testing - Martin Fowler

https://martinfowler.com/articles/2021-test-shapes.html

When to Mock - Robert C. Martin (Uncle Bob)

https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html

Additional resources

TDD, Where Did It All Go Wrong - Ian Cooper

https://youtu.be/EZ05e7EMOLM

Is TDD dead? - Martin Fowler, Kent Beck, DHH

https://youtu.be/z9quxZsLcfo

unit-testing-diet GitHub repository

https://github.com/steliosfran/unit-testing-diet

Do not mock every layer

Test the behavior, not the implementation

“Eat food. Not too much. Mostly plants.”

“Write tests. Not too much. Mostly on the ViewModel.”