Swift Unit Testing Reimagined: Fresh Perspectives

Daniel Zhang
July 7, 2023

Contents

Unit testing might seem like a chore — just a bunch of tedious code for verifying other code, right? But what if we could shift our perspective and see it in a new light? Drawing from a wide range of experiences and an openness to fresh ideas, we can approach unit testing from a different angle. In this article, I'll show you how unit testing can be as thrilling and full of potential as embarking on a brand-new greenfield project.

1 Evolving Test-Driven Development: A New Approach

I have a confession to make. My approach to test-driven development (TDD) and unit testing has been incomplete. Yes, using tests to guide and develop code is fundamental, but there's so much more to it. I'm talking about levels of abstraction that go beyond just writing and passing tests.

In our latest project, we've taken a more holistic approach. Of course, testing still drives the creation of production code as it should in TDD. But we've also started using tests to inform project decisions involving dependencies and data privacy. Our criteria for success extend beyond just having tests. We're investing in dependency isolation and creating a synthetic but deterministic testing data environment.

As a result, our testing process has become about more than ensuring code correctness. It's now a key part of our software design and architecture process, helping us to create cleaner, more maintainable code.

1.1 Lessons Learned from Deployment: The Value of Synthetic Data Environments

In past projects, we have learned many lessons from delivering code to millions of devices. A synthetic data environment, illustrated in Figure 1, is one of our arsenal's most potent tools. It's a testing environment that's not just a clone of the production one. It's where we can see how things work in a playground setting, enjoying the freedom of dependency isolation and abstraction. Instead of running our production pipeline fully, we can choose the components or functionalities we want to test. And we can reshape the data environment to suit our needs.

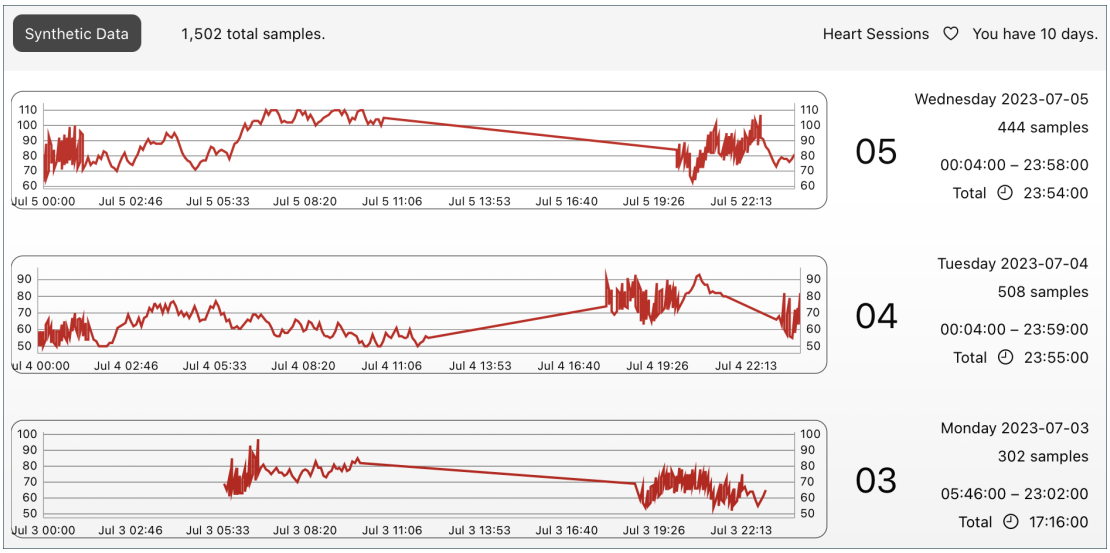

Figure 1: Synthetic data environment. Reading the midnight boundary crossing between July 04 and 05 confirms a five-minute bin interval, according to the time difference between 23:59 and 00:04.

1.2 Prioritizing Data Privacy and Collaboration: The Role of the Testing Environment

Why go through so much trouble to create a synthetic testing environment? For our application, it makes absolute sense because we deal with private health data. By having it, we prioritize data privacy, streamline testing efforts, and foster effective collaboration and communication among team members while increasing their flexibility and control over development.

- **Data Privacy**: By creating a separate testing data environment, we can ensure that private or sensitive data is not exposed during testing. This allows contributors to test their code and share examples without compromising data privacy. It fosters a secure and trusted testing environment, promoting collaboration among team members.
- **Clearer Testing and Collaboration**: The synthetic testing environment allows us to test parts of the production pipeline selectively. Instead of running the entire pipeline, we can choose the components or function-

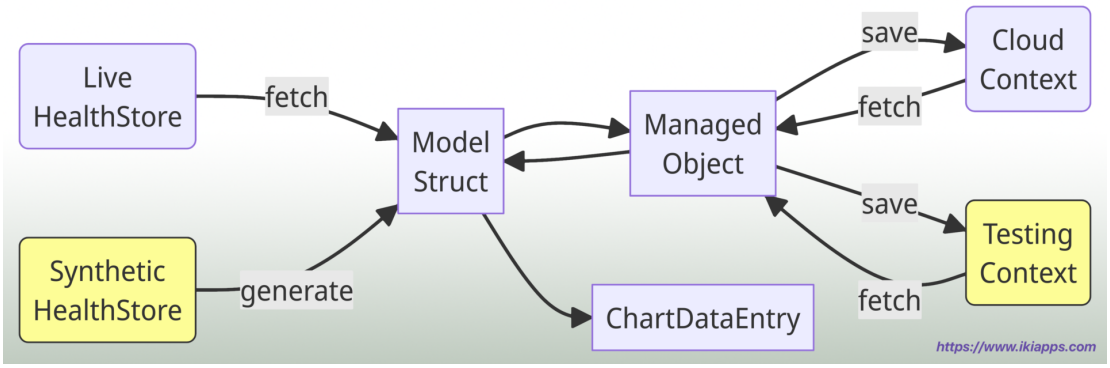

Figure 2: System design data flow diagram.

alities we want to test. This targeted testing approach streamlines the testing process and makes it more efficient. Team members can focus on testing relevant aspects and providing feedback, enhancing collaboration and communication.

- **Flexibility and Control**: We can reshape the data environment to suit various needs. We can manipulate the data to simulate various scenarios and edge cases, which helps uncover potential issues. This flexibility and control promote effective communication within the team, as they can discuss and analyze different data scenarios and their impact on the system.

2 A Practical Example of Abstracting Dependencies and Isolating the Data Environment

Where do we draw the line on abstraction? It's a complex question, but it's worth exploring with an example straight from our project. Our system design diagram, Figure 2, shows the data flow from a health store, like one accessed by HealthKit, to a managed object context (MOC) and how we process data for charts. It can be seen how the synthetic testing environment works interchangeably with the production environment.

2.1 Synthesizing a Health Store

We provide a synthetic health store by using a protocol that abstracts behaviors on the concrete health store. In this case, it is not a direct replacement for HKHealthStore.

```
protocol HealthStoreProtocol
{
    var healthStore: HKHealthStore? { get set }
    func authorizePublisher() -> AnyPublisher<Bool, Error>
    func heartRateQueryPublisher()
        -> AnyPublisher<QuantitySampleProtocol, Error>
}
```

Another protocol, QuantitySampleProtocol, abstracts the concrete HKQuantitySample and serves as a data model for our synthetic data.

```
protocol QuantitySampleProtocol
{
    var count          : Int { get }
    var device         : HKDevice? { get }
    var endDate        : Date { get }
    var metadata       : [String: Any]? { get }
    var quantity       : HKQuantity { get }
    var sourceRevision: HKSourceRevision? { get }
    var startDate      : Date { get }
    var uuid           : UUID { get }
}
```

The ability to generate synthetic data simply by specifying the number of records we want to create is a powerful tool. It is the next step in the evolution of our testing process because it offers new ways to test our system with deterministic data flows beyond just passing tests with static data.

2.2 Abstracting Dependencies with Combine

Using Combine as a foundational framework for our testing environment makes it easy to inject abstract dependencies. The following code shows an example of creating a synthetic data publisher along with a mock, concrete implementation of the QuantitySampleProtocol.

```
/// - Returns: A publisher that emits a QuantitySampleProtocol.
func heartRateQueryPublisher()
    -> AnyPublisher<QuantitySampleProtocol, Error>
{
    let quantitySamples: [QuantitySampleProtocol] = MockHeartRateData
        .dummyHeartRateSamples(numberOfSamples: numberOfRecords)
        .map
        { (sample: HeartRateSample) in
            let quantity = HKQuantity(
                unit: HKUnit(from: "count/min"),
                doubleValue: sample.bpm
            )
            return QuantitySampleProtocolMock(
                quantity: quantity, startDate: sample.startDate,
                endDate: sample.endDate, metadata: [:], uuid: UUID(),
                sourceRevision: nil, device: nil, count: 1
            )
        }
    return quantitySamples.publisher
        .setFailureType(to: Error.self)
        .eraseToAnyPublisher()
}
```

By using protocols, we can selectively inject dependencies. The approach is flexible and allows us to isolate different concerns for testing, production, or some combination of both.

2.3 Unit Testing with the Synthetic Data Environment

The subsequent test illustrates our application of the synthetic data environment for code verification, with a couple takeaways.

- **Building Block:** The synchronous version presented herein serves as the foundation for our multithreaded Core Data operations.
- **Streamlined Process:** The key insight is that the utilization of a synthetic data environment has significantly streamlined the process of constructing and testing new functionalities.

```
/// Save a targetRecordCount number of records to a managed context.
func test_save_records_to_context() throws
{
    let expectation = expectation(description: #function)
    let sut = MockHealthStore(numberOfRecords: targetRecordCount)
    var received = 0
    _ = sut.authorizePublisher()
        .flatMap
        { _ -> AnyPublisher<QuantitySampleProtocol, Error> in
            sut.heartRateQueryPublisher()
        }
        .sink
        { (_: Subscribers.Completion<Error>) in
            expectation.fulfill()
        } receiveValue: { (value: QuantitySampleProtocol) in
            let sample = value.toSample() // Convert to a model struct.
            do
            {
                try sample.saveToCoreData(managedContext: self.context)
                received += 1
            } catch
            {
                XCTFail(error.localizedDescription)
```

```
            }
        }
    wait(for: [expectation], timeout: timeout)
    XCTAssertEqual(received, targetRecordCount)
}
```

We use the SUT (System Under Test) notation to identify and represent the system actively being tested. Adopting this tiny convention gives every test a built-in template for communicating adherence to the isolation principle.

The integration of a synthetic testing environment blurs the boundaries between testing and production within our system's design.

This interchangeable nature empowers us to address production issues while conducting tests under controlled conditions. We gain the ability to toggle various system components, isolating and thoroughly testing specific elements. Additionally, we can simulate diverse scenarios and edge cases to gauge the system's resilience.

By leveraging protocols to abstract dependencies, a natural dependency inversion occurs, contributing to the creation of a more resilient and robust system design.

Please see our article on Combating Technical Debt with the Power of Dependency Inversion in Swift for our introduction to dependency inversion.

3 Final Thoughts: Building Robust Systems with Synthetic Testing Environments

In summary, we have seen how a synthetic testing environment can be a powerful tool for testing and development.

1. **Unit Testing Approach:** Unit testing can be approached differently, emphasizing holistic development and design guidance.

2. **Synthetic Data Environment:** A synthetic data environment, different from the production environment, allows for selective testing and manipulation of data.
3. **Data Privacy and Collaboration:** Creating a separate testing data environment prioritizes data privacy and fosters collaboration, streamlining testing efforts.
4. **Flexibility of Synthetic Testing Environment:** The synthetic testing environment provides flexibility, control, and the ability to simulate various scenarios and edge cases.
5. **Abstraction through Protocols:** Abstraction through protocols helps isolate dependencies with the dependency inversion principle, creating a highly flexible data environment unbounded by production constraints.
6. **Generating Synthetic Data:** The ability to generate synthetic data offers new ways to test code deterministically beyond just passing tests with static data.
7. **Development Approach:** Approaching development from a testing and dependency inversion perspective helps create a more robust and resilient system that is easier to maintain.

3.1 Performance Benefits

We did not have time to cover the performance benefits of the approach to testing and simulation we have described. However, we plan to address this topic in a future article.

Our synthetic testing environment has directly contributed to advancements in our data processing pipeline and UI performance. Our technology stack is Swift, Combine, Core Data, HealthKit, CloudKit, and SwiftUI. We have reduced space and time complexity by factors of 100x for some primary, production use cases through studies conducted with synthetic data.

A more comprehensive description will require detailing the specific testing

methods, the nature of the data, and the exact conditions under which the results were achieved. So many valuable insights have been gained, and we are excited to share them with you.

3.2 Feedback

We value your feedback on our article and would greatly appreciate your input as we continuously strive to improve and provide valuable content. Your thoughts and comments are important to us, and we thank you in advance for taking the time to share them with us!