Harnessing The Power of

Hexagonal Architecture

Note: Hyperlinks are not active when viewed from GitHub's file viewer. Either navigate back to the main ReadMe for the project and click the link for this document in the Design Notes section or type this link into your browser ***https://woodyb.github.io/vending-machine-project/design/Harnessing-The-Power-of-Hexagonal-Architecture.pdf***

Benefits of Hexagonal Architecture

Hexagonal architecture, with its emphasis on testability and modularization, stands out as a powerful design pattern. However, it's essential to recognize that hexagonal architecture need not exist in isolation – It can be seamlessly integrated with other architectural paradigms to unlock even greater benefits. It’s not an all or nothing decision when you add hexagonal architecture.

Hexagonal Architecture: A Foundation for Testability

Hexagonal architecture, also known as **ports and adapters** architecture, provides a solid foundation for building software systems that are both flexible and testable. By decoupling application logic from external dependencies, it enables thorough testing and promotes independent development of components. The benefits of hexagonal architecture extend beyond testability to include flexibility, separation of concerns, and enhanced collaboration.

Embracing Integration

While hexagonal architecture offers numerous advantages on its own, it can also be combined with other architectural patterns to amplify its benefits. Integrating hexagonal architecture with microservices, event-driven architecture, or serverless architecture to name a few enables teams to achieve a more versatile and powerful software design.

Microservices Architecture

Pairing hexagonal architecture with microservices facilitates the development of independently deployable services with modularized internal components. This combination enhances agility and reliability while ensuring testability and flexibility at the architectural level. Having worked before as a SDET responsible for testing microservices I often found there was some testing that I wanted to conduct that was very difficult because I could not isolate the core logic in the microservice. Some ports and adapters in the

microservices would have made this work so much easier. What I've come to realize over the years is you don't need SDETs that can perform heroics to find ways to test your product if you are creating testable software, which will be better designed software.

Event-Driven Architecture

In event-driven systems, hexagonal architecture can be applied within individual components to ensure testability and independence from infrastructure concerns. This approach enables the development of resilient, scalable systems that can evolve over time while benefiting from the isolation provided by hexagonal architecture.

Serverless Architecture

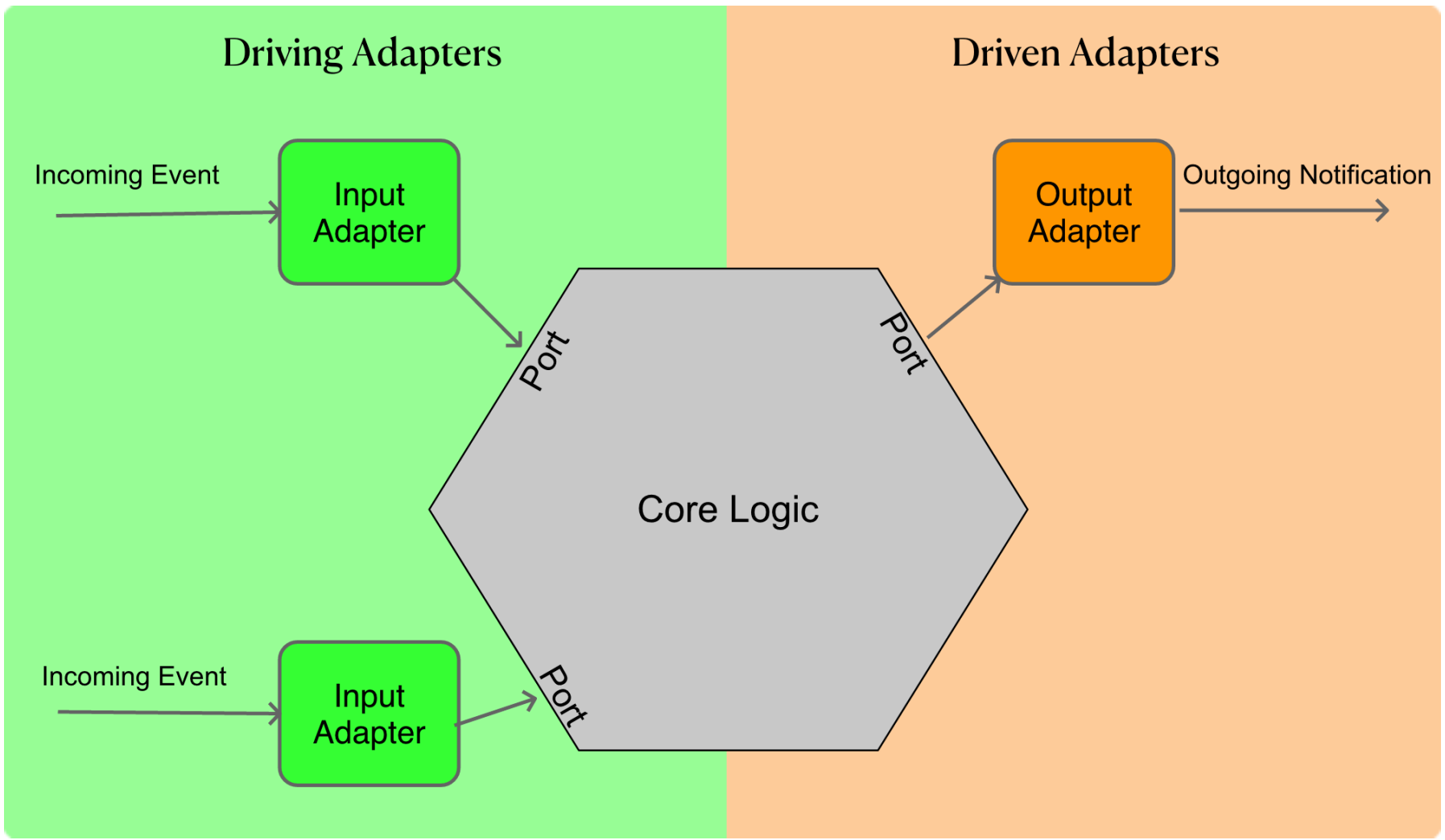
Combining hexagonal architecture with serverless functions enhances modularity and testability within serverless applications. By encapsulating business logic and dependencies, developers can achieve greater agility and reliability in their serverless deployments.

Embracing A Hybrid Approach

The integration of hexagonal architecture with other architectural patterns represents a hybrid approach that leverages the strengths of each paradigm. By embracing this approach, teams can unlock the full potential of their software designs, achieving greater flexibility, maintainability, and testability in an ever-changing technological landscape.

I recently discovered Tech Excellence led by Valentina Cupać. There are a lot of good videos there concerning topics like TDD in Hexagonal Architecture and Clean Architecture.

Hexagonal Architecture In The Vending Machine Project



Basic Idea

I'll leave it to you to dig in on hexagonal architecture but at a high level it looks something like this. Here's a link to Dr. Alistair Cockburn's website which is a good place to start.

The core logic is inside the hexagon and only interacts with outside entities through a port. What's a port in this context? It's an agreed upon way for the core logic to communicate with an adapter. In software terms this will usually be an interface if the language supports them. We are using Typescript and it does support Interfaces. So we can think Port equals Interface for this project.

Now, what are adapters? Adapters are kind of glue code. Driving adapters take incoming events and convert the information as necessary in order to send it through a port to the core logic. Code-wise, in this project these will normally be implemented as classes. Driven adapters just do the inverse in that they take output from the core logic and convert it so it can be used by whatever is connected to the other side of the adapter.

Vending Machine Logic Controller

The Logic Controller is our core logic. It is being implemented as a FSM (Finite State Machine) for now. However, if we decide to implement the core logic another way as long as we don't change the ports (Interfaces) our tests should still function.

Port Example

Let's look at an example of a port and adapter in our project. Here's an input port for getting the coins the user has put into the vending machine before making a selection. I don't know exactly how real vending machines work but I do know they have something called a coin mechanism. The core logic has to have some way to find out if the user has inserted enough money to make a selection. I've created this interface to make it easy for the core logic to get the information it needs. It will be the adapter's job to interact with a coin mechanism however it works so that the core logic can consume it through the port. In our case the core logic will be polling `readInsertedCoin()`. The other method `insertCoin()` will be called by our simulator when the user uses the keyboard to simulate inserting a coin. Example: pressing the q key to insert a quarter. This port is a contract between the core logic and the outside world. The idea is no matter how the data comes in as long as this contract is in place the core logic doesn't care. It's the adapter's job to handle how to communicate with the outside world to get the information abstracting the core logic from the particulars.

```
export interface CoinMechanismInsertedCoinsInterface {
    insertCoin(coin: Coins): void;
    readInsertedCoin(): Coins;
}
```

Adapter Example

Ok so we've got a port, how about the adapter. We don't have a real coin mechanism but we do have a simulator in this project. The vending machine software along with the simulator is our shipping "product." The simulator is being created to allow a user to interact with the vending machine in ways like inserting coins. The core logic has no idea of what's on the other side of the interface nor should it. Let's have a look at two adapters that implement the `CoinMechanismInsertedCoinsInterface`.

1) CoinMechanismInsertedCoinsSimulatorAdapter

2) MockCoinMechanismInsertedCoinsAdapter in VendingMachine.spec.ts

My hope is the long names give you a good clue to what the other side of the adapter is connected to. The first one is our simulator and the second one is being used for unit tests when we are testing the core logic. In this project often adapters used by tests are just implemented in the test code and not in an individual file.

Since both of these adapters implement the `CoinMechanismInsertedCoinsInterface` they must provide methods as described in the interface or they will not compile. However, that's not the only methods they may implement. We can augment the class to have additional methods. This allows us to create a test doubles which can be a stubs, mocks, spies etc.

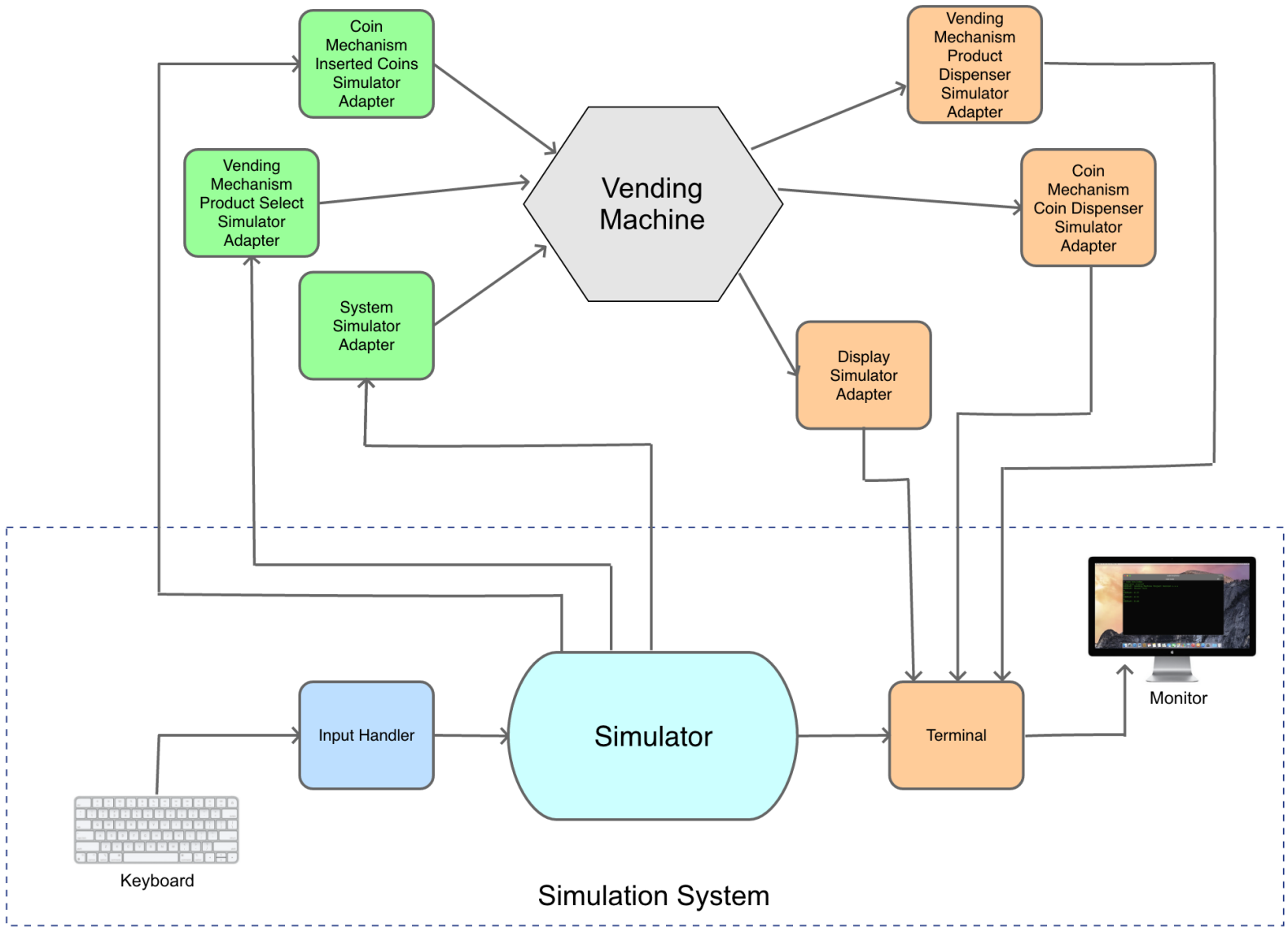
`CoinMechanismInsertedCoinsSimulatorAdapter` is sent coins (quarters, dimes, nickels) from our simulator and holds the value until the vending machine logic can read them.

`MockCoinMechanismInsertedCoinsUnitTestAdapter` exists so tests can control what the what the adapter will return to the core logic when it calls `readInsertedCoin().`

Having complete control over all the adapters (control inputs / capture outputs) allows us to completely control the core logic's environment and thoroughly test it. To see more of these tests have a look at the commit test VendingMachine.spec.ts

The Vending Machine With Simulator Diagram

Note: This diagram is a work in progress and will be updated as functionality is added to the Vending Machine. You may need to refresh your browser a few times in order to see the most up to date version.



Why All This Abstraction?

Couldn't this be much simpler just to demonstrate the necessary logic and tests for the Vending Machine code kata? Yes of course but that's not the whole goal of the project. If you look at our diagram above we could completely throw away our simulator, replace the simulator adapters with new adapters that interface with real hardware components and not change our core logic in Vending Machine. This separation of concerns not only allows us to move the core logic to a whole new world with high confidence it also allows us to throughly test it since we can control its inputs and capture its outputs. If you look at the

unit test for the Vending Machine (commit test VendingMachine.spec.ts) you will see that's exactly what we are doing. We also have separate tests for our Simulator so it can be tested independently of the Vending Machine.