KMAC: State Machine Template

Technical Architecture Document.

Author: @huitemagico

1. Summary:

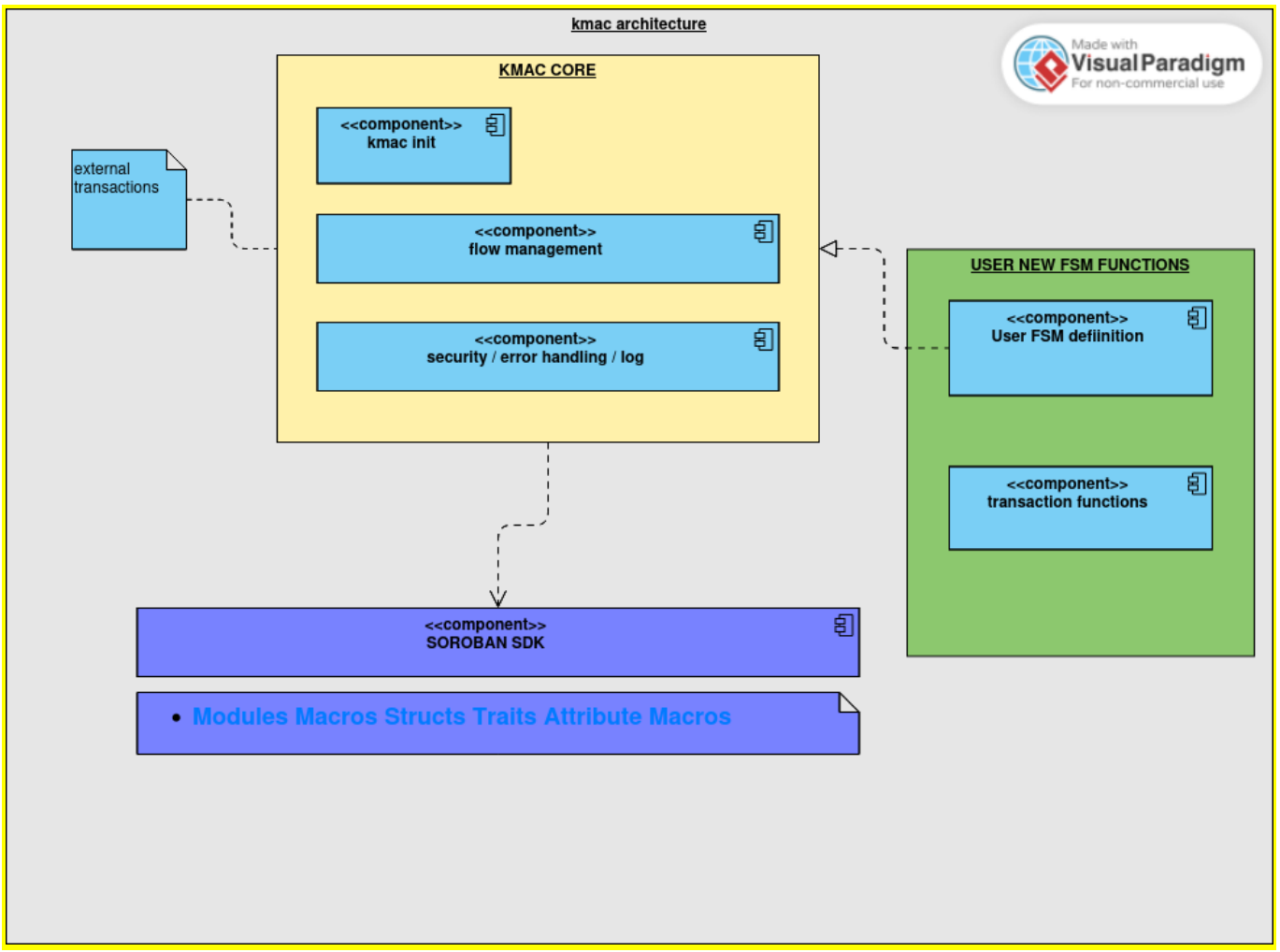
The term “State Machine” or "**Finite State Machine (FSM)**" represents a mathematical model employed in various computing domains, including the design and development of **Smart Contracts**.

The **KMAC** project proposes a smart contract program constructed with the **soroban-sdk**, which implements a Finite State Machine template, leveraging well-established design patterns, and offering extensible functional capabilities.

The product is a functional application that provides a template for creating an FSM-style process with additional functionalities. This is possible because KMAC has a “core” component, including a base for flow management, along with security functions. The core follows **standard design patterns**.

In this way, you can easily add new functions to the template, create a new process, and reuse the core components.

Fig.1 KMAC Architecture



2. Functionalities:

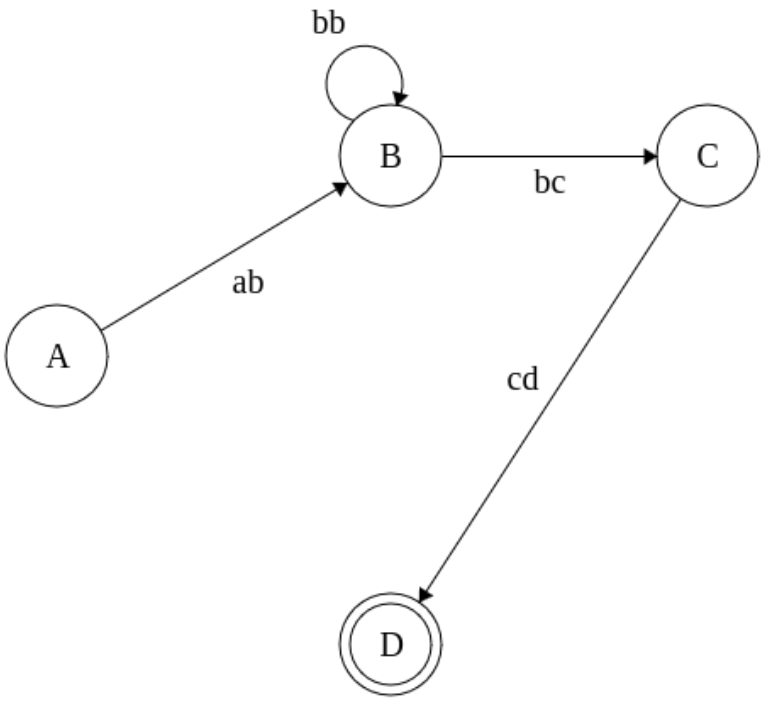
From a functional perspective, KMAC allows the implementation of "finite state machine" functionalities, which entail a flow composed of a set of states and transactions. The functionality of the FSM to be constructed is defined in part by:

- The state and transaction transition table.
- A state transition diagram.
- The specification of transactions. (*)

(*) Please refer to the figures “Fig. 2” and “Fig. 3” for more details.

In any given state (state X), the program can only accept a valid transaction, as defined by the transition table. Based on this transaction and, potentially, data associated with the progress made so far, it can change the flow's state to a new one.

Fig.2 FSM diagram



The diagram above (Fig.2) shows an FSM defined for the states A,B,C,D, and transactions “ab”,”bb”,”bc”,”cd”.

State A is the initial state, state D is the ending state.

The transaction “ab” allows that the state of the machine flows from state A to state B.

The transaction “bb” maintains the state of the machine in state B.

The transaction “bc” allows that the state of the machine flows from B to C.

The transaction “cd” allows that the state of the machine flows from C to D.

As D is the ending state, one transaction “da” would return the machine to an initial state (not shown in the diagram).

Fig.3 FSM Transition Table

The table below is another view of the same process shown in Figure 2.

	A	**B**	**C**	**D**
A		**ab**		
B		**bb**	**bc**	
C				**cd**
D				

Each transaction would implement functions as updating data of the FSM instance, that has to be persisted, because each transaction could get the environment data before the transaction.

3. Roles, Actors, Functions

KMAC defines different users and their roles like this:

1. Developer Role: Developers use the Soroban SDK to create the KMAC project. Once the production version is out, their job is to keep it updated and running smoothly.
2. Builder Role: This role involves adding new features to the Finite State Machine (FSM). Initially, the developer may also be the builder. But, another user who wants to add a new process can take the KMAC template and work on coding the transaction functions and data settings to build that new process.
3. Business User Role: These users have specific needs that can be met by creating a new system using the FSM approach based on KMAC. They are the ones with requirements that the builder user can fulfill by using the KMAC template.

4. Lifecycle of an Application Generated with the KMAC Template

The process of creating a KMAC application, which helps make a new process that works like an FSM, starts when a business user explains what they need and outlines the steps their process will go through. The user-builder then sets up the initial data and defines how the flow should work. After that, they write the code for the specific functions their application needs. These functions have to follow a specific set of rules.

The basic structure of how the FSM works, along with the security and error handling, is already set up in the "core" part of KMAC. The user-builder only needs to code the transaction functions.

This way, KMAC makes it possible to add more functions to each step in the process. So, the KMAC model allows for creating an FSM program with new functions defined by the business user and coded by the user-builder.

5. Why is important

KMAC, the FSM template, can help the Stellar and Soroban developer community grow. It provides a strong foundation and methods that make things easier for developers and users.

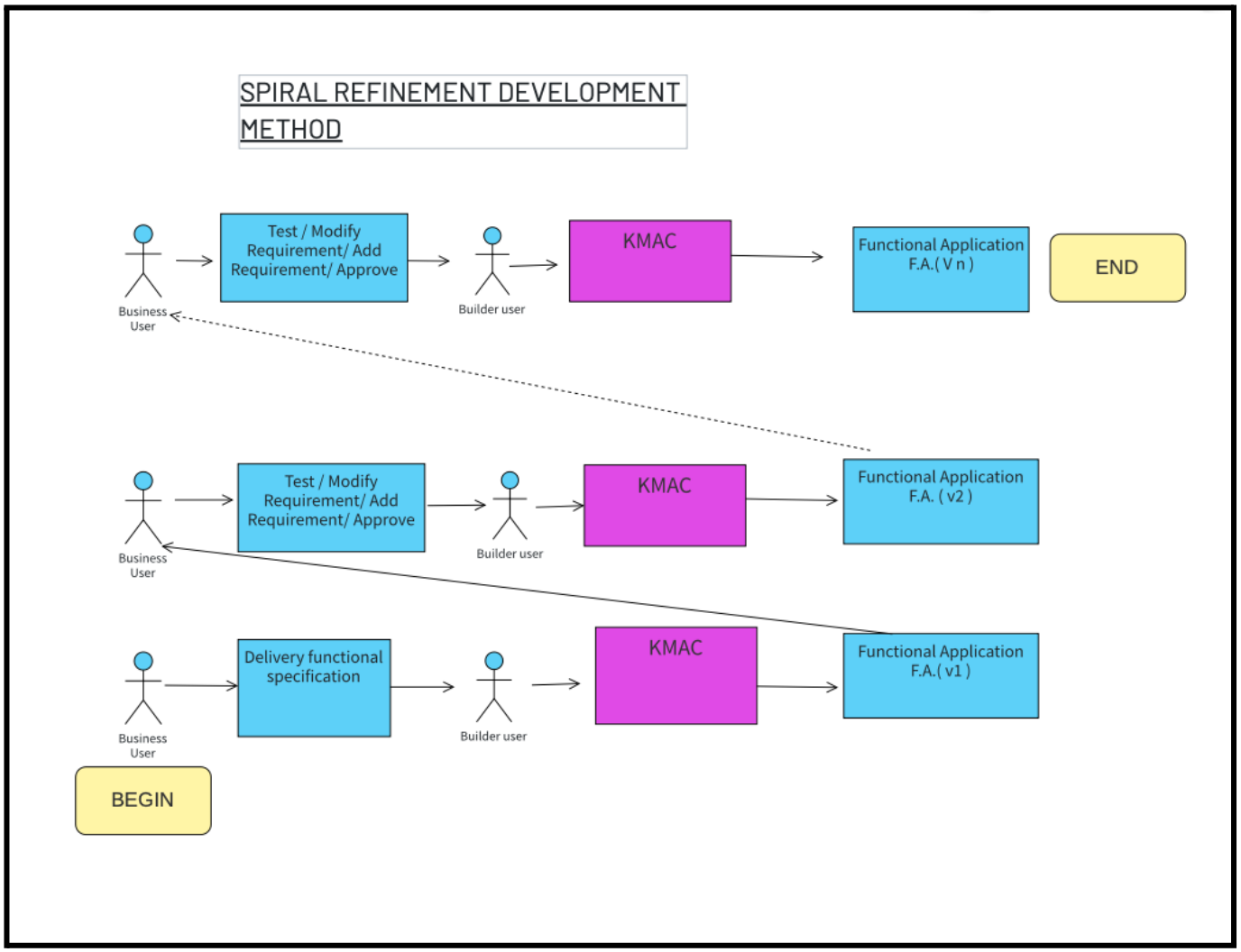
Here's why KMAC matters for our community:

- **Implementation of Design Patterns:** For **developers**, KMAC's modular setup makes it easy to **study, test, and document design patterns in Smart Contracts** created with Soroban SDK. KMAC handles things like security, access control, avoiding repeated work, and fixing errors.
- Additionally, it's an experimental tool to assess the effectiveness and efficiency of these design patterns' implementation.
- From a developer's perspective, as KMAC adheres to a **modular design**, it allows for **understanding, exploring, testing, and documenting** the design patterns of Smart Contracts implemented with the Soroban SDK.

- **Flexibility in Design:** KMAC encourages a flexible setup, where the user-builder defines transaction-related functions, making Smart Contract development more versatile.

- **Rapid Delivery for End Users:** KMAC can be used as an “Spiral Refinement Development Method” (see Fig.4). In this way, end users can gradually and progressively test their requirements, allowing the final product to align precisely with their needs, increasing the likelihood of avoiding post-production adjustments.

Fig. 4 Spiral Refinement Development Method



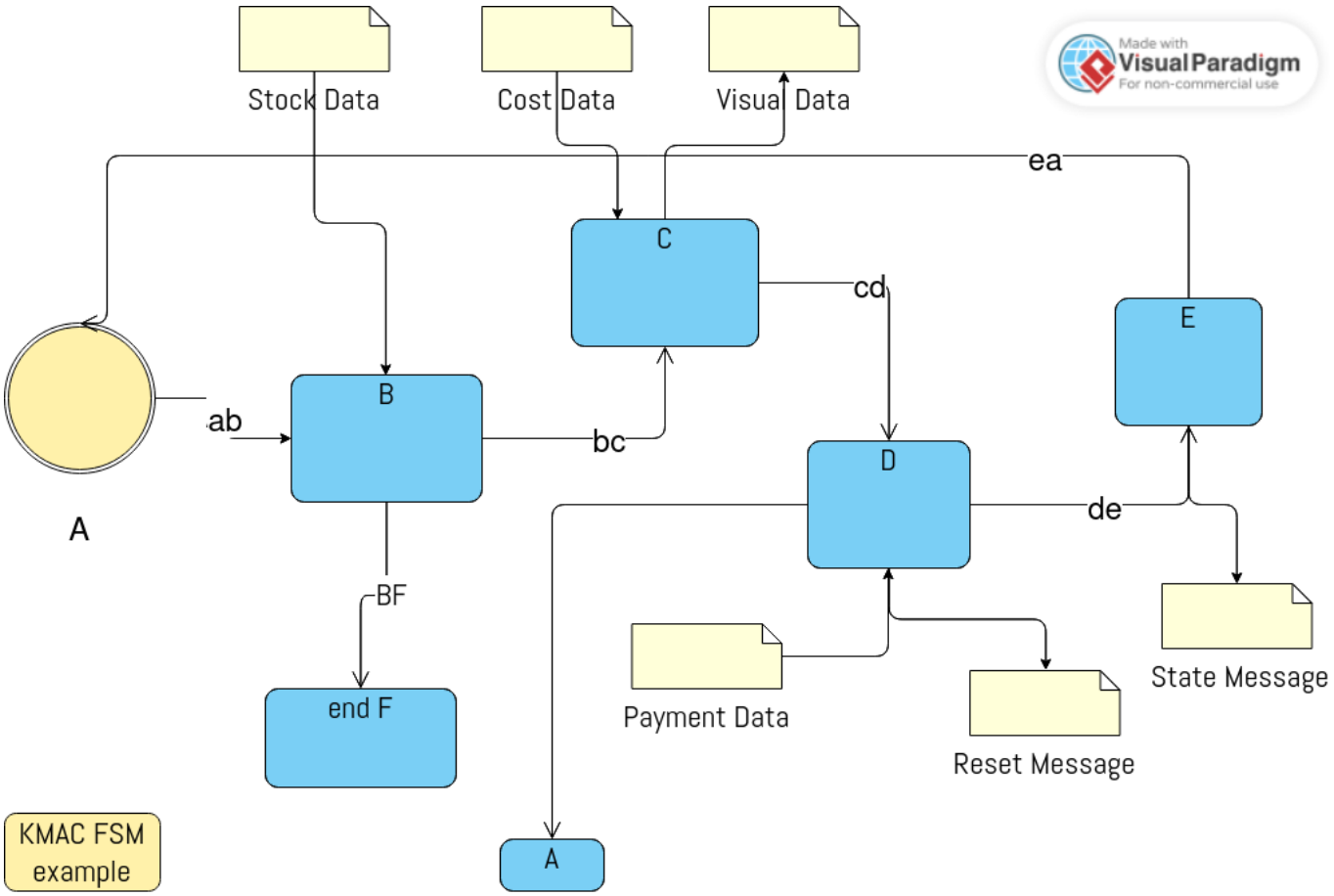
6. **Vending machine Example:**

Let's consider a **practical example**: a coffee vending machine. Initially, the requirement is the following: the machine should control the mixture of coffee components, accept payment, and dispense the beverage.

Here's how it works: (please see Fig. 5)

- In State A (Machine Ready): The machine is turned on and ready to take an order.
- In State A (User Selects Beverage)(transaction “ab”): The user picks a drink option (in this case, there's only one choice). This takes us to State B.
- In State B (Component Stock Check): The machine checks if it has all the ingredients it needs. If it's running low, it goes to a failure state (State F) (transaction “bf”) . If it has enough, it moves to State C. (transaction “bc”)
- In State C (Payment Calculation): The machine calculates the price and shows it. We assume the user pays the exact amount. Then it goes to State D. (Internal transaction “cd”)
- In State D (Waiting for Payment): If the user pays, it moves to State E. (internal transaction “de”). If not, it goes back to the initial State A. (internal transaction “da”)
- In State E (Beverage Dispensing): The machine makes the coffee and serves it. If everything goes well, it goes back to the initial State A, (internal transaction “ea”) ready for the next customer.
- In State F (Failure State): This is when the machine runs out of supplies or encounters a problem. It stays in this state until something not shown in the diagram takes it back to the initial state.

Fig. 5 FSM diagram for the Vending Machine example



7. **Challenges and limitations of KMAC:**

(A) Data Management: Dealing with information like time, user details, and machine-specific data within the process can be complicated. The current version of KMAC assumes that in the future, there will be external data sources (like an oracle) that will be updated through transaction functions. For example, when the machine needs new supplies, the data source is updated.

In the deliverable version, it uses hard-coded data in memory for input data and doesn't update the output data.

(B) Memory Efficiency: Managing a large amount of data in the contract's permanent memory can use up a lot of resources. KMAC in future versions could manage this by allowing parameterization based on machine identification. It can fetch external data from the data source using these parameters, making the contract work with different machines.

However, in the deliverable version, this is only discussed in the documentation, and there's no specific implementation.