Dynamic State Management: Composable State Machines with Primitive Operations

Introduction

State machines are a proven approach to managing complex system behavior. This article presents a state machine implementation pattern that emphasizes code reusability through composable primitive operations and the Chain of Responsibility pattern. While traditional state machines often intertwine state logic with behavior, this approach separates primitive operations from state definitions, allowing for high reusability and simplified maintenance.

The core concept centers on building states from chains of reusable primitive operations. New system behaviors can be created by composing existing primitives in different configurations, with minimal need for new code. When new functionality is required, developers need only implement new primitive operations, which then become available for use across all states.

Architectural Components

Primitive Operations

Primitive operations form the foundational building blocks of the system. Each primitive:

- Performs a single, well-defined operation
- Returns a success/failure status
- Is stateless and independently testable
- Can be chained with other primitives
- Is registered in a central primitive registry
- Contributes to and uses a shared execution context

Example primitive signature in Go:

```go
type ExecutionContext struct {
    Data map[string]interface{}
}

type PrimitiveResult struct {
    Success   bool
    NextState string
    Data      map[string]interface{}
}

type Primitive interface {
    Execute(context *ExecutionContext) (*PrimitiveResult, error)
}
```

State Definitions

States are defined through configuration rather than code, therefore, need to be properly created by reading the configuration from a database or a repository. Each state consists of:

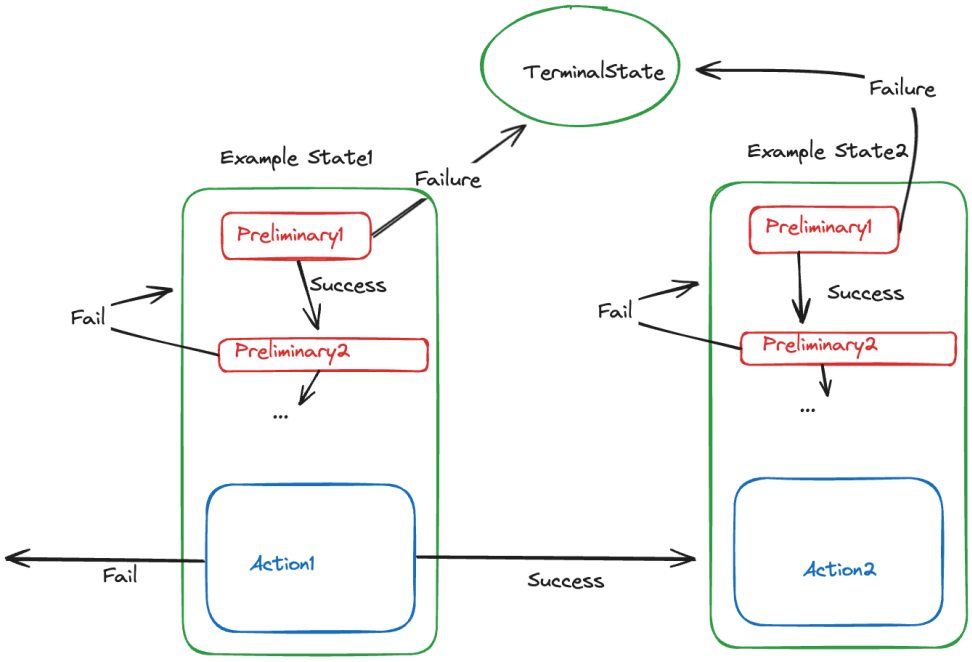
- A set of preliminary actions (ordered primitive chains)
- A main action (primary primitive)
- Success and failure transition rules

Example state configuration in Go:

```go
type NextState string
```

```
type StateDefinition struct {
    Name              string
    PreliminaryActions []PrimitiveChain
    MainAction        string
    Transitions       struct {
        Success NextState
        Failure NextState
    }
}

type PrimitiveChain struct {
    Primitives    []string
    ExecutionOrder int
}
```



A picture that speaks louder than the blog itself

Chain of Responsibility Implementation

The system implements the Chain of Responsibility pattern at two levels:

Level 1: Primitive Chaining

Preliminary actions form chains of primitive operations:

```
type PrimitiveChainExecutor struct {
    PrimitiveRegistry map[string]Primitive
}

func (pce *PrimitiveChainExecutor) Execute(chain PrimitiveChain, context *ExecutionContext) (*PrimitiveResult, error) {
```

```
    for _, primitiveName := range chain.Primitives {
        primitive, exists := pce.PrimitiveRegistry[primitiveName]
        if !exists {
            return nil, fmt.Errorf("primitive not found: %s", primitiveName)
        }

        result, err := primitive.Execute(context)
        if err != nil {
            return nil, err
        }

        if !result.Success {
            return result, nil // Break chain on failure
        }

        // Update context with result data
        for k, v := range result.Data {
            context.Data[k] = v
        }
    }
    return &PrimitiveResult{Success: true}, nil
}
```

Level 2: State Execution

States execute their preliminary action chains in order, followed by the main action:

```
type StateExecutor struct {
    StateDefinitions map[string]StateDefinition
    ChainExecutor    PrimitiveChainExecutor
}

func (se *StateExecutor) ExecuteState(stateName string, context *ExecutionContext) (string, error) {
    state, exists := se.StateDefinitions[stateName]
    if !exists {
        return "", fmt.Errorf("state not found: %s", stateName)
    }

    // Execute preliminary actions
    for _, chain := range state.PreliminaryActions {
        result, err := se.ChainExecutor.Execute(chain, context)
        if err != nil {
            return "", err
        }
        if !result.Success {
            return state.Transitions.Failure, nil
        }
    }

    // Execute main action
    mainPrimitive, exists := se.ChainExecutor.PrimitiveRegistry[state.MainAction]
    if !exists {
        return "", fmt.Errorf("main action not found: %s", state.MainAction)
    }

    mainResult, err := mainPrimitive.Execute(context)
    if err != nil {
        return "", err
```

```
    }

    if mainResult.Success {
        return state.Transitions.Success, nil
    } else {
        return state.Transitions.Failure, nil
    }
}
```

Context Maintenance

A crucial aspect of this system is the maintenance of a shared execution context. Each primitive operation both contributes to and utilizes this context, allowing data to flow between primitives and states. The `ExecutionContext` struct serves as a container for all relevant data throughout the state machine's execution.

The main action of each state can access the accumulated data from preliminary actions through this context, enabling complex decision-making based on the collective results of multiple primitives.

UI-Based Configuration

This approach to state machine design lends itself well to UI-based configuration. A graphical interface can be developed that allows domain experts to:

1. Define new states
2. Configure primitive chains for each state
3. Set up transitions between states
4. Modify existing state configurations

This UI can generate the necessary configuration files or database entries, enabling rapid iteration on business processes without requiring developer intervention for every change.

Domain-Specific Language Specifications

The configuration-driven nature of this system opens up possibilities for creating domain-specific languages (DSLs) to define state machines. A DSL can provide a more intuitive and concise way to express state definitions, primitive chains, and transitions.

Example DSL specification:

```
STATE: OrderReceived
  PRELIM:
    - ValidateOrder
    - CheckInventory
  MAIN: ProcessPayment
  SUCCESS -> OrderFulfillment
  FAILURE -> OrderCancelled

STATE: OrderFulfillment
  PRELIM:
    - AllocateInventory
    - GenerateShippingLabel
  MAIN: ShipOrder
  SUCCESS -> OrderCompleted
  FAILURE -> CustomerServiceReview
```

This DSL can be parsed and translated into the underlying Go structures, providing a user-friendly way to define complex state machines.

Benefits of This Approach

Reusability

- Primitives are independent, reusable units
- The same primitive can be used in multiple states
- Primitive chains can be reused across states

Maintainability

- Clear separation between state logic and primitive operations
- Primitives can be tested in isolation
- State flows can be modified through configuration
- Easier tracing and logging

Extensibility

- New primitives can be added without modifying existing code
- States can be created or modified using any combination of primitives
- Primitive chains can be extended or modified without code changes

Best Practices

1. Primitive Design
   - Keep primitives focused on a single responsibility
   - Ensure primitives are stateless
   - Design primitives to be context-independent
2. State Configuration
   - Group related primitives in logical chains
   - Keep preliminary action chains focused
   - Consider chain order carefully
3. Error Handling
   - Define clear failure paths
   - Implement proper rollback mechanisms
   - Maintain execution context for debugging

Conclusion

This approach to state machine design emphasizes code reusability through composable primitive operations, implemented in Go. By maintaining a shared execution context and enabling UI-based configuration, it bridges the gap between technical implementation and business process management. The potential for domain-specific language specifications further enhances its flexibility and usability.

While it requires careful design of primitive operations and thoughtful management of the shared context, the benefits in terms of maintainability, extensibility, and business-IT alignment are significant. As systems grow in complexity, this pattern provides a scalable way to manage intricate business processes while minimizing code changes.

The key to success lies in designing primitives that are truly reusable, and maintaining a clear separation between primitive operations and state definitions. When new functionality is required, developers should first consider how existing primitives can be composed to achieve the desired behavior, implementing new primitives only when necessary.

By adopting this pattern, organizations can create more adaptable systems that evolve with changing business needs, fostering collaboration between technical teams and domain experts. The result is a more agile, maintainable, and extensible approach to managing complex state-driven processes.