🔄 Gno vs Solidity

Smart Contract Patterns for Solidity Developers

Bridging the gap between Ethereum and Gno.land

👋 Who is this for?

Target Audience:

- **Experienced Solidity Developers**
- Ethereum smart contract developers
- Web3 developers looking to expand to Gno.land
- Anyone familiar with EVM-based development

What you'll learn:

- How Gno compares to Solidity
- Common patterns translated from Solidity to Gno
- Best practices for Gno development

🌐 What is Gno?

Gno is a blockchain platform using **Go-based smart contracts**

Key Features:

- **Language**: Based on Go (Golang)
- **Paradigm**: Interpreted, not compiled to bytecode
- **Type System**: Strong static typing
- **Concurrency**: Deterministic (no goroutines in contracts)
- **Storage**: Persistent state with realms

Philosophy:

Write smart contracts in a **familiar, readable language** (Go) instead of learning Solidity from scratch

⚔️ Gno vs Solidity: Language

Feature	Solidity	Gno
Base Language	Custom	Go (Golang)
Execution	Compiled to bytecode	Interpreted
Type System	Static	Static (Go-based)
Inheritance	Yes (OOP)	No (composition)
Interfaces	Yes	Yes (Go interfaces)
Error Handling	`require`, `revert`	`panic`, `error` returns
Gas Model	EVM gas	Gno gas

📦 Pattern 1: Basic Counter

Solidity

```
contract Counter {
    uint256 private count;

    function increment() public {
        count += 1;
    }

    function getCount() public view returns (uint256) {
        return count;
    }
}
```

Gno

```
package counter

var count int

func Increment() {
  count++
}

func GetCount() int {
  return count
}
```

🔒 Pattern 2: Ownable / Access Control

Solidity

```
contract Ownable {
    address private owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    function restrictedAction() public onlyOwner {
        // Only owner can call
    }
}
```

Gno

```
package ownable

import "chain/runtime"

var owner address

func init() {
  owner = runtime.PreviousRealm().Addr()
}

func AssertIsOwner() {
  if runtime.PreviousRealm().Addr() != owner {
    panic("not owner")
  }
}

func RestrictedAction() error {
  AssertIsOwner()
  // Only owner can call
  return nil
}
```

🔐 Pattern 3: State Lock

Solidity

```
contract StateLock {
    enum State { Open, Closed, Paused }
    State public state;

    modifier inState(State _state) {
        require(state == _state, "Invalid state");
        _;
    }

    function doSomething() public inState(State.Open) {
        // Only callable when Open
    }
}
```

Gno

```
package statelock

type State int

const (
  Open State = iota
  Closed
  Paused
)

var state State = Open

func AssertState(expected State) {
  if state != expected {
    panic("invalid state")
  }
}

func DoSomething() {
  AssertState(Open)
  // Only callable when Open
}
```

📣 Pattern 4: Emit Events

Solidity

```
contract EventEmitter {
    event Transfer(address indexed from, address indexed to,

    function transfer(address to, uint256 amount) public {
        // Transfer logic
        emit Transfer(msg.sender, to, amount);
    }
}
```

Gno

```
package events

import (
  "chain/runtime"

  "gno.land/p/demo/emit"
)

func Transfer(to address, amount uint64) {
  // Transfer logic
  emit.Emit("Transfer",
    "from", runtime.PreviousRealm().Addr().String(),
    "to", to.String(),
    "amount", amount,
  )
}
```

🏭 Pattern 5: Factory Pattern

Solidity

```
contract Factory {
    address[] public contracts;

    function createContract() public returns (address) {
        MyContract newContract = new MyContract();
        contracts.push(address(newContract));
        return address(newContract);
    }
}
```

Gno

```
package factory

import "gno.land/p/demo/avl"

var contracts avl.Tree // realm_path -> instance

func CreateContract(name string) string {
  // Create new realm dynamically
  path := "gno.land/r/demo/" + name
  // Note: Dynamic realm creation not yet supported
  // Use sub-packages or registry pattern instead
  contracts.Set(path, &Contract{})
  return path
}
```

💰 Pattern 6: Token (ERC20 → GRC20)

Solidity (ERC20)

```
contract ERC20 {
    mapping(address => uint256) public balanceOf;

    function transfer(address to, uint256 amount) public {
        require(balanceOf[msg.sender] >= amount);
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
    }
}
```

Gno (GRC20)

```
package grc20

import (
  "chain/runtime"

  "gno.land/p/demo/grc/grc20"
)

var banker *grc20.Banker

func Transfer(to address, amount uint64) error {
  caller := runtime.PreviousRealm().Addr()
  return banker.Transfer(caller, to, amount)
}
```

🛡️ Pattern 7: Reentrancy Guard

Solidity

```solidity
contract ReentrancyGuard {
    bool private locked;

    modifier noReentrant() {
        require(!locked, "Reentrant call");
        locked = true;
        _;
        locked = false;
    }

    function withdraw() public noReentrant {
        // Safe withdrawal
    }
}
```

Gno

```gno
package reentrancy

var locked bool

func AssertNotLocked() {
  if locked {
    panic("reentrant call")
  }
  locked = true
}

func Unlock() {
  locked = false
}

func Withdraw() {
  AssertNotLocked()
  defer Unlock()
  // Safe withdrawal
}
```

💸 Pattern 8: Wallet - Withdraw

Solidity

```solidity
contract Wallet {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount);
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }
}
```

Gno

```gno
package wallet

import (
  "chain"
  "chain/runtime"

  "gno.land/p/demo/avl"
)

var balances avl.Tree // Address -> uint64

func Withdraw(amount uint64) {
  caller := runtime.PreviousRealm().Addr()
  balance := getBalance(caller)
  if balance < amount {
    panic("insufficient balance")
  }
  setBalance(caller, balance-amount)
  banker.SendCoins(caller, chain.Coins{{amount, "ugnot"}})
}
```

💸 Pattern 9: Wallet - Deposit

Solidity

```solidity
contract Wallet {
    mapping(address => uint256) public balances;

    receive() external payable {
        balances[msg.sender] += msg.value;
    }

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
}
```

Gno

```gno
package wallet

import (
  "chain/banker"
  "chain/runtime"
)

func Deposit() {
  caller := runtime.PreviousRealm().Addr()
  sent := banker.OriginSend()

  if len(sent) == 0 {
    panic("no coins sent")
  }

  amount := sent[0].Amount
  balance := getBalance(caller)
  setBalance(caller, balance+amount)
}
```

🗺️ Pattern 10: Mapping (AVL Tree)

Solidity

```
contract Mapping {
    mapping(address => uint256) public balances;
    mapping(string => address) public registry;

    function set(string memory key, address value) public {
        registry[key] = value;
    }
}
```

Gno

```
package mapping

import "gno.land/p/demo/avl"

var balances avl.Tree // Address -> uint64
var registry avl.Tree // string -> Address

func Set(key string, value address) {
  registry.Set(key, value)
}

func Get(key string) address {
  val, _ := registry.Get(key)
  return val.(address)
}
```

✅ Pattern 11: Guard Check

Solidity

```
contract GuardCheck {
    modifier validAmount(uint256 amount) {
        require(amount > 0, "Amount must be positive");
        require(amount ⩽ 1000, "Amount too large");
        _;
    }

    function transfer(uint256 amount) public validAmount(amo
        // Transfer logic
    }
}
```

Gno

```
package guardcheck

func AssertValidAmount(amount uint64) {
  if amount == 0 {
    panic("amount must be positive")
  }
  if amount > 1000 {
    panic("amount too large")
  }
}

func Transfer(amount uint64) {
  AssertValidAmount(amount)
  // Transfer logic
}
```

🔄 Pattern 12: Proxy Pattern

Solidity

```
contract Proxy {
    address public implementation;

    function upgrade(address newImpl) public onlyOwner {
        implementation = newImpl;
    }

    fallback() external payable {
        address impl = implementation;
        assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), impl, 0, callc
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

Gno

```
package proxy

import "gno.land/p/demo/ownable"

var implementation string // realm path

func Upgrade(newImpl string) {
  ownable.AssertCallerIsOwner()
  implementation = newImpl
}

func Call(method string, args ...interface{}) interface{}
  // Delegate to implementation realm
  // Note: Direct delegation not yet supported
  // Use interface-based composition instead
  return callRealm(implementation, method, args...)
}

// Alternative: Use DAO pattern like GovDAO
// with UpdateImpl() to switch implementations
```

🚨 Pattern 13: Emergency Stop

Solidity

```
contract EmergencyStop {
    bool public stopped = false;

    modifier stopInEmergency() {
        require(!stopped, "Contract stopped");
        _;
    }

    function emergencyStop() public onlyOwner {
        stopped = true;
    }

    function resume() public onlyOwner {
        stopped = false;
    }
}
```

Gno

```
package emergency

import "gno.land/p/demo/ownable"

var stopped bool = false

func AssertNotStopped() {
  if stopped {
    panic("contract stopped")
  }
}

func EmergencyStop() {
  ownable.AssertCallerIsOwner()
  stopped = true
}

func Resume() {
  ownable.AssertCallerIsOwner()
  stopped = false
}

func DoSomething() {
  AssertNotStopped()
  // Normal logic
}
```

⏱️ Pattern 14: Speed Bump

Solidity

```
contract SpeedBump {
    mapping(address ⇒ uint256) public lastActionTime;
    uint256 public constant DELAY = 1 days;

    modifier rateLimit() {
        require(block.timestamp ⩾ lastActionTime[msg.sender
        _;
        lastActionTime[msg.sender] = block.timestamp;
    }

    function withdraw() public rateLimit {
        // Withdrawal logic
    }
}
```

Gno

```
package speedbump

import (
  "chain/runtime"
  "time"

  "gno.land/p/demo/avl"
)

var lastActionTime avl.Tree // Address -> time.Time
const DELAY = 24 * time.Hour

func AssertRateLimit() {
  caller := runtime.PreviousRealm().Addr()
  lastTime, exists := lastActionTime.Get(caller.String())

  now := time.Now()
  if exists && now.Sub(lastTime.(time.Time)) < DELAY {
    panic("rate limit: please wait")
  }
  lastActionTime.Set(caller.String(), now)
}

func Withdraw() {
  AssertRateLimit()
  // Withdrawal logic
}
```

🗝️ Key Differences Summary

Solidity	Gno
`msg.sender`	`std.PrevRealm().Addr()`
`require()`	`if !condition { panic() }`
`modifier`	Function calls with `Assert*()`
`mapping`	`avl.Tree`
`emit Event`	`emit.Emit()`
`contract`	`package` + `realm`
`constructor`	`init()` function
`payable`	`std.GetOrigSend()`

🎯 Best Practices for Gno

Coming from Solidity:

1. **Think packages, not contracts** - Gno uses Go packages
2. **Use composition over inheritance** - No OOP inheritance
3. **Leverage Go's standard library** - Familiar Go patterns work
4. **AVL trees for mappings** - Efficient key-value storage
5. **Panic for errors** - Or return `error` types
6. **Realms are stateful** - Persistent state across calls
7. **No dynamic dispatch** - Static linking at deploy time

📚 Resources

Learning More:

- **Gno Documentation**: docs.gno.land
- **Gno by Example**: gno.land/r/demo
- **Standard Library**: `gno.land/p/demo/*`
- **Example Realms**: `gno.land/r/demo/*`

Key Packages:

- `gno.land/p/demo/avl` - AVL tree (like mapping)
- `gno.land/p/demo/ownable` - Ownership pattern
- `gno.land/p/demo/grc/grc20` - Token standard

🚀 Getting Started

Next Steps:

1. **Set up Gno development environment**
2. **Clone and explore example realms**
3. **Start with simple counter or blog**
4. **Gradually implement familiar patterns**
5. **Join the Gno community**

Remember:

- Gno is **Go** - if you know Go, you're 80% there
- Patterns are **similar** - just different syntax
- Community is **helpful** - ask questions!