Introduction

While there have been other resource based programming languages in the past (such as Plaid), for a long time resource based programming has been a niche, lacking applications where resource based approach would excel. However, smart contracts and blockchains introduced software development challenges that can be answered with resource based programming.

The Move language and its Move Virtual Machine were designed originally by Facebook to power their Diem blockchain platform providing an intuitive environment and ecosystem for resource oriented applications. Fortunately Move was adopted by multiple blockchain projects as their smart contract execution environment before Diem was discontinued, so Move continues living separately as an independent project.

Today Move and its virtual machine are powering multiple blockchains, most of which are still in the early development phase.

Because Move is currently the most widely developed resource based programming language for blockchains, code examples are written in Move. However it is likely that some of these patterns can be implemented also in some other resource based language and ecosystem.

What this book is

This book is for discussing software design paradigms and best practices for resource based languages, especially Move and its flavors.

What this book is not

This book is not a guide to Move or any other resource based language. For books on Move itself, see this list. Also see awesome-move for a curated list of code and content from the Move programming language community.

Technical disclaimer

This book is designed to be viewed digitally with hyperlink support (such as PDF or web format). For now the full software pattern format is not followed, instead the patterns are simply defined by a short summary and examples.

License

Capability

Name	Capability
Origin	Libra Project / Unknown Author
Example	Sui Move by Example / Damir Shamanaev
Depends on	None
Known to work on	Move

Summary

A **Capability** is a resource proving that the owner of the resource is permitted to execute a certain action. This is the oldest known Move design pattern dating back to the Libra project and its token smart contract, where capabilities were used to authorize minting of coins.

Examples

```
module examples::item {
    use sui::transfer;
    use sui::id::VersionedID;
    use sui::utf8::{Self, String};
    use sui::tx_context::{Self, TxContext};

    /// Type that marks Capability to create new `Item`s.
    struct AdminCap has key { id: VersionedID }

    /// Custom NFT-like type.
    struct Item has key, store { id: VersionedID, name: String }

    /// Module initializer is called once on module publish.
    /// Here we create only one instance of `AdminCap` and send it to the publisher.
    fun init(ctx: &mut TxContext) {
        transfer::transfer(AdminCap {
            id: tx_context::new_id(ctx)
        }, tx_context::sender(ctx))
    }

    /// The entry function can not be called if `AdminCap` is not passed as
    /// the first argument. Hence only owner of the `AdminCap` can perform
    /// this action.
    public entry fun create_and_send(
        _: &AdminCap, name: vector<u8>, to: address, ctx: &mut TxContext
    ) {
        transfer::transfer(Item {
            id: tx_context::new_id(ctx),
            name: utf8::string_unsafe(name)
        }, to)
    }
}
```

Example for Sui Move is taken from the book Sui Move by Example by Damir Shamanaev.

Multiple Configurations

Name	Multiple Configurations
Origin	Ville Sundell
Example	Diem Forum Post
Depends on	None
Known to work on	Move

Summary

In the **Multiple Configurations** approach singletons (especially for configuration data) are provided to the module as arguments instead of storing them to any specific location.

By incorporating this design into your modules, you would automatically add a layer of configurability by giving users or admins (if any) a way to create configurations which others could use. This would be similar to Uniswap which lets anyone create markets others could participate in.

Examples

Let's say you have an auction smart contract X and an admin has created a configuration to account A. Then B could issue a transaction script to bid on C's resource like this:

```
X::place_bid(A, C, 1234);
```

By accepting parameters (such as C and the amount 1234), a transaction script could trivially hide the selection of the configuration. This depends, how flexible the final script implementation is in terms of arguments to `main()`, but the transaction script could look like this (quick mock-up):

```
script {
    use 0xFF::X;

    const CONF: address = 0xCOFFEE;

    fun main(a: address, amount: XUS) {
        X::place_bid(CONF, a, amount);
    }
}
```

Nestable Resources

Name	Nestable Resources
Origin	Ville Sundell
Example	TaoHe project
Depends on	None
Known to work on	Move

Summary

Nestable Resources pattern provides a resource native approach to code reuse by building a nested structure of resources where each resource can be detached for use if the conditions are satisfied. For example, tokens can be placed into a time locked resource, which in turn can be placed into a resource which can be used only by a certain user, effectively creating a timelocked non-fungible token. For cleaner design and easier integration it is recommended that the nestable resources share a common interface for wrapping and unwrapping.

Examples

```
/// A nested resource (tao) for implementing a simple ownership model: owner can
extract
/// the content.
module 0x1::Ownable {
    use Std::Signer;

    /// Simple ownership tao: the `owner` can extract `content`.
    struct Tao<Content> has key, store {
        owner: address,
        content: Content
    }

    /// Wrapping `content` into a tao the `owner` can only extract.
    public fun wrap<Content>(owner: address, content: Content): Tao<Content> {
        Tao<Content> { owner, content }
    }

    /// Immutable read-only reference to the owner address, and `content`.
    public fun read<Content>(tao: &Tao<Content>): (&address, &Content) {
        let Tao<Content> { owner, content } = tao;

        (owner, content)
    }

    /// If `account ` is the `owner`, extract `content`.
    public fun unwrap<Content>(account: &signer, tao: Tao<Content>): Content {
        let Tao<Content> { owner, content } = tao;

        assert!(owner == Signer::address_of(account), 123);

        content
    }
}
```

```
/// A folder tao to store an arbitrary number of taos.
module 0x1::Folder {
    /// A simple tao struct containing a vector of resources.
    struct Tao<Content> has key, store {
        content: vector<Content>
    }

    /// Create a new tao, with the static set of resources inside it.
    public fun wrap<Content>(content: vector<Content>): Tao<Content> {
        Tao<Content> { content }
    }

    /// Immutable read-only reference to the vector containing resources.
    public fun read<Content>(tao: &Tao<Content>): &vector<Content> {
        let Tao<Content> { content } = tao;

        content
    }

    /// Destroy the tao, and return the static set of resources inside it.
    public fun unwrap<Content>(tao: Tao<Content>): vector<Content> {
        let Tao<Content> { content } = tao;

        content
    }
```

A token can be placed inside `Ownable` and it can be placed inside `Folder`, or other way around, if so desired.

Witness

Name	Witness
Origin	FastX / Sam Blackshear
Example	Sui Move by Example / Damir Shamanaev
Depends on	None
Known to work on	Move

Summary

A **Witness** is an ephemeral resource designed to prove only once that the a resource in question can be initiated only once after the witness has been created. The resource is dropped after use, ensuring that the same resource cannot be reused to initialize any other struct.

Examples

```
/// Module that defines a generic type `Guardian<T>` which can only be
/// instantiated with a witness.
module examples::guardian {
    use sui::id::VersionedID;
    use sui::tx_context::{Self, TxContext};

    /// Phantom parameter T can only be initialized in the `create_guardian`
    /// function. But the types passed here must have `drop`.
    struct Guardian<phantom T: drop> has key, store {
        id: VersionedID
    }

    /// The first argument of this function is an actual instance of the
    /// type T with `drop` ability. It is dropped as soon as received.
    public fun create_guardian<T: drop>(
        _witness: T, ctx: &mut TxContext
    ): Guardian<T> {
        Guardian { id: tx_context::new_id(ctx) }
    }
}

/// Custom module that makes use of the `guardian`.
module examples::peace {
    use sui::transfer;
    use sui::tx_context::{Self, TxContext};

    // Use the `guardian` as a dependency.
    use 0x0::guardian;

    /// This type is intended to be used only once.
    struct PEACE has drop {}

    /// Module initializer is the best way to ensure that the
    /// code is called only once. With `Witness` pattern it is
    /// often the best practice.
    fun init(ctx: &mut TxContext) {
```

```
        transfer::transfer(
            guardian::create_guardian(PEACE {}, ctx),
            tx_context::sender(ctx)
        )
    }
}
```

Example for Sui Move is taken from the book Sui Move by Example by Damir Shamanaev.

Accountless Design

Name	Accountless Design
Origin	Ville Sundell
Example	Diem Forum Post
Depends on	None
Known to work on	Move

Summary

Move module following the **Accountless Design** pattern doesn't handle storage (`move_to()` / `move_from()`) directly, instead the storage must be handled outside the module in transaction scripts. This makes the module code footprint smaller, design simpler, implementation more portable and provides a way to implement storage agnostic smart contract design on some Move powered platforms.

Examples

```
module 0x1::Outbox {
    use Std::Event;
    use Std::Signer;
    use Std::Vector;

    struct Item<Content: key + store> has key, store {
        from: address,
        to: address,
        content: Content
    }

    struct Outbox<Content: key + store> has key, store {
        content: vector<Item<Content>>
    }

    struct Put<phantom Content> has key, drop, store {
    }

    struct EventHandle<phantom Content: drop + store> has key, store {
        event_handle: Event::EventHandle<Content>
    }

    public fun create<Content: key + store>(account: &signer) {
        move_to<Outbox<Content>>(account, Outbox<Content> { content:
Vector::empty<Item<Content>>() });
        move_to<EventHandle<Put<Content>>>(account, EventHandle<Put<Content>> {
event_handle: Event::new_event_handle<Put<Content>>(account) } );
    }

    public fun put<Content: key + store>(account: &signer, from: address, to: address,
content: Content) acquires EventHandle, Outbox {
        let outbox_owner = Signer::address_of(account);
        let event_handle = borrow_global_mut<EventHandle<Put<Content>>>(outbox_owner);
        let outbox = borrow_global_mut<Outbox<Content>>(outbox_owner);

        assert!(to != @0x0, 123);
```

```
        Vector::push_back<Item<Content>>(&mut outbox.content, Item<Content>{ from, to,
content });
        Event::emit_event<Put<Content>>(&mut event_handle.event_handle, Put<Content>
{});
    }

    public fun get<Content: key + store>(account: &signer, outbox_owner: address,
index: u64): Content acquires Outbox {
        let account_addr = Signer::address_of(account);
        let outbox = borrow_global_mut<Outbox<Content>>(outbox_owner);

        let Item<Content>{from, to, content} = Vector::swap_remove<Item<Content>>(&mut
outbox.content, index);

        assert!(from == account_addr || to == account_addr, 123);

        content
    }
}
```

Now `Outbox` can be used to retrieve and store resources in transaction scripts, and pass those to modules following the Script Based Design pattern.

Hot Potato

Name	Hot Potato
Origin	Sui Project / Todd Nowacki
Example	FlashLender.move
Depends on	None
Known to work on	Move

Summary

A **Hot Potato** is a struct without `key`, `store` and `drop` abilities forcing the struct to be used within the transaction it was created in. This is desired in applications like flash loans where the loans must be initiated and repaid during the same transaction.

Examples

```
// Copyright (c) 2022, Mysten Labs, Inc.
// SPDX-License-Identifier: Apache-2.0

/// A flash loan that works for any Coin type
module defi::flash_lender {
    use sui::balance::{Self, Balance};
    use sui::coin::{Self, Coin};
    use sui::object::{Self, ID, UID};
    use sui::transfer;
    use sui::tx_context::{Self, TxContext};

    /// A shared object offering flash loans to any buyer willing to pay `fee`.
    struct FlashLender<phantom T> has key {
        id: UID,
        /// Coins available to be lent to prospective borrowers
        to_lend: Balance<T>,
        /// Number of `Coin<T>`'s that will be charged for the loan.
        /// In practice, this would probably be a percentage, but
        /// we use a flat fee here for simplicity.
        fee: u64,
    }

    /// A "hot potato" struct recording the number of `Coin<T>`'s that
    /// were borrowed. Because this struct does not have the `key` or
    /// `store` ability, it cannot be transferred or otherwise placed in
    /// persistent storage. Because it does not have the `drop` ability,
    /// it cannot be discarded. Thus, the only way to get rid of this
    /// struct is to call `repay` sometime during the transaction that created it,
    /// which is exactly what we want from a flash loan.
    struct Receipt<phantom T> {
        /// ID of the flash lender object the debt holder borrowed from
        flash_lender_id: ID,
        /// Total amount of funds the borrower must repay: amount borrowed + the fee
        repay_amount: u64
    }
```

```
    /// An object conveying the privilege to withdraw funds from and deposit funds to
the
    /// `FlashLender` instance with ID `flash_lender_id`. Initially granted to the
creator
    /// of the `FlashLender`, and only one `AdminCap` per lender exists.
    struct AdminCap has key, store {
        id: UID,
        flash_lender_id: ID,
    }

    /// Attempted to borrow more than the `FlashLender` has.
    /// Try borrowing a smaller amount.
    const ELoanTooLarge: u64 = 0;

    /// Tried to repay an amount other than `repay_amount` (i.e., the amount borrowed +
the fee).
    /// Try repaying the proper amount.
    const EInvalidRepaymentAmount: u64 = 1;

    /// Attempted to repay a `FlashLender` that was not the source of this particular
debt.
    /// Try repaying the correct lender.
    const ERepayToWrongLender: u64 = 2;

    /// Attempted to perform an admin-only operation without valid permissions
    /// Try using the correct `AdminCap`
    const EAdminOnly: u64 = 3;

    /// Attempted to withdraw more than the `FlashLender` has.
    /// Try withdrawing a smaller amount.
    const EWithdrawTooLarge: u64 = 4;

    // === Creating a flash lender ===

    /// Create a shared `FlashLender` object that makes `to_lend` available for
borrowing.
    /// Any borrower will need to repay the borrowed amount and `fee` by the end of the
    /// current transaction.
    public fun new<T>(to_lend: Balance<T>, fee: u64, ctx: &mut TxContext): AdminCap {
```

```
        let id = object::new(ctx);
        let flash_lender_id = object::uid_to_inner(&id);
        let flash_lender = FlashLender { id, to_lend, fee };
        // make the `FlashLender` a shared object so anyone can request loans
        transfer::share_object(flash_lender);

        // give the creator admin permissions
        AdminCap { id: object::new(ctx), flash_lender_id }
    }

    /// Same as `new`, but transfer `WithdrawCap` to the transaction sender
    public entry fun create<T>(to_lend: Coin<T>, fee: u64, ctx: &mut TxContext) {
        let balance = coin::into_balance(to_lend);
        let withdraw_cap = new(balance, fee, ctx);

        transfer::transfer(withdraw_cap, tx_context::sender(ctx))
    }

    // === Core functionality: requesting a loan and repaying it ===

    /// Request a loan of `amount` from `lender`. The returned `Receipt<T>` "hot
potato" ensures
    /// that the borrower will call `repay(lender, ...)` later on in this tx.
    /// Aborts if `amount` is greater that the amount that `lender` has available for
lending.
    public fun loan<T>(
        self: &mut FlashLender<T>, amount: u64, ctx: &mut TxContext
    ): (Coin<T>, Receipt<T>) {
        let to_lend = &mut self.to_lend;
        assert!(balance::value(to_lend) >= amount, ELoanTooLarge);
        let loan = coin::take(to_lend, amount, ctx);
        let repay_amount = amount + self.fee;
        let receipt = Receipt { flash_lender_id: object::id(self), repay_amount };

        (loan, receipt)
    }

    /// Repay the loan recorded by `receipt` to `lender` with `payment`.
    /// Aborts if the repayment amount is incorrect or `lender` is not the
```

```
`FlashLender`
    /// that issued the original loan.
    public fun repay<T>(self: &mut FlashLender<T>, payment: Coin<T>, receipt:
Receipt<T>) {
        let Receipt { flash_lender_id, repay_amount } = receipt;
        assert!(object::id(self) == flash_lender_id, ERepayToWrongLender);
        assert!(coin::value(&payment) == repay_amount, EInvalidRepaymentAmount);

        coin::put(&mut self.to_lend, payment)
    }

    // === Admin-only functionality ===

    /// Allow admin for `self` to withdraw funds.
    public fun withdraw<T>(
        self: &mut FlashLender<T>,
        admin_cap: &AdminCap,
        amount: u64,
        ctx: &mut TxContext
    ): Coin<T> {
        // only the holder of the `AdminCap` for `self` can withdraw funds
        check_admin(self, admin_cap);

        let to_lend = &mut self.to_lend;
        assert!(balance::value(to_lend) >= amount, EWithdrawTooLarge);
        coin::take(to_lend, amount, ctx)
    }

    /// Allow admin to add more funds to `self`
    public entry fun deposit<T>(
        self: &mut FlashLender<T>, admin_cap: &AdminCap, coin: Coin<T>
    ) {
        // only the holder of the `AdminCap` for `self` can deposit funds
        check_admin(self, admin_cap);
        coin::put(&mut self.to_lend, coin);
    }

    /// Allow admin to update the fee for `self`
    public entry fun update_fee<T>(
```

```
        self: &mut FlashLender<T>, admin_cap: &AdminCap, new_fee: u64
    ) {
        // only the holder of the `AdminCap` for `self` can update the fee
        check_admin(self, admin_cap);

        self.fee = new_fee
    }

    fun check_admin<T>(self: &FlashLender<T>, admin_cap: &AdminCap) {
        assert!(object::borrow_id(self) == &admin_cap.flash_lender_id, EAdminOnly);
    }

    // === Reads ===

    /// Return the current fee for `self`
    public fun fee<T>(self: &FlashLender<T>): u64 {
        self.fee
    }

    /// Return the maximum amount available for borrowing
    public fun max_loan<T>(self: &FlashLender<T>): u64 {
        balance::value(&self.to_lend)
    }

    /// Return the amount that the holder of `self` must repay
    public fun repay_amount<T>(self: &Receipt<T>): u64 {
        self.repay_amount
    }

    /// Return the amount that the holder of `self` must repay
    public fun flash_lender_id<T>(self: &Receipt<T>): ID {
        self.flash_lender_id
    }
}
```

Example for Sui Move is taken from the Sui repository.

Script Based Design

Name	Script Based Design
Origin	Ville Sundell
Example	Diem Forum post lost, April 18 2022
Depends on	Accountless Design
Known to work on	Move

Summary

Script Based Design combines Accountless Design with *Move Transaction Scripts* enabling a design where most of the business logic resides in transaction scripts, while keeping the most critical parts in modules. This way the transaction scripts can be developed and improved faster than a regular module would, while providing the same kind of guarantees as regular modules by keeping the most critical part (state transitions) in modules.

Examples

Transferable Witness

Name	Transferable Witness
Origin	Sui Move by Example / Damir Shamanaev
Example	Sui Move by Example / Damir Shamanaev
Depends on	Capability, Witness
Known to work on	Move

Summary

A **Transferable Witness** is a semi-ephemeral storable witness wrapped into a disposable capability.

Examples

```
/// This pattern is based on combination of two others: Capability and a Witness.
/// Since Witness is something to be careful with, spawning it should be only
/// allowed to authorized users (ideally only once). But some scenarios require
/// type authorization by module X to be used in another module Y. Or, possibly,
/// there's a case where authorization should be performed after some time.
///
/// For these, rather rare, scerarios a storable witness is a perfect solution.
module examples::transferable_witness {
    use sui::transfer;
    use sui::id::{Self, VersionedID};
    use sui::tx_context::{Self, TxContext};

    /// Witness now has a `store` which allows us to store it inside a wrapper.
    struct WITNESS has store, drop {}

    /// Carries the witness type. Can only be used once to get a Witness.
    struct WitnessCarrier has key { id: VersionedID, witness: WITNESS }

    /// Send a `WitnessCarrier` to the module publisher.
    fun init(ctx: &mut TxContext) {
        transfer::transfer(
            WitnessCarrier { id: tx_context::new_id(ctx), witness: WITNESS {} },
            tx_context::sender(ctx)
        )
    }

    /// Unwrap a carrier and get the inner WITNESS type.
    public fun get_witness(carrier: WitnessCarrier): WITNESS {
        let WitnessCarrier { id, witness } = carrier;
        id::delete(id);
        witness
    }
}
```

Example for Sui Move is taken from the book Sui Move by Example by Damir Shamanaev.

ID Pointer

Name	ID Pointer
Origin	Sui Project
Example	example.move
Depends on	None
Known to work on	Move

Summary

The following example implements basic `Lock` and `Key` mechanics on Sui where `Lock<T>` is a shared object that can contain any object, and `Key` is an owned object, which is required to get access to the contents of the lock.

`Key` is linked to its `Lock` using an `ID` field. This check allows off-chain discovery of the target, as well as splits the dynamic transferable capability and the 'static' contents. Another benefit of this approach is that the target asset is always discoverable, while you can wrap its `Key` into another object (such as a marketplace listing).

ID pointer is a technique that separates the main data (an object) and its accessors/capabilities by linking the latter to the original. There's a few different ways you can use this pattern:

- Issuing transferable capabilities for shared objects (for example, a TransferCap that changes 'owner' field of a shared object)
- Splitting dynamic data and static (for example, an NFT and its collection information)

- Avoiding unnecessary type linking (and witness requirement) in generic applications (LP token for a `LiquidityPool`)

Examples

The following example implements basic Lock and Key mechanics on Sui where Lock is a shared object that can contain any object, and Key is an owned object, which is required to get access to the contents of the lock.

Key is linked to its Lock using an ID field. This check allows off-chain discovery of the target, as well as splits the dynamic transferable capability and the 'static' contents. Another benefit of this approach is that the target asset is always discoverable, while you can wrap its Key into another object (such as a marketplace listing).

```
// Copyright (c) 2022, Mysten Labs, Inc.
// SPDX-License-Identifier: Apache-2.0

module examples::lock_and_key {
    use sui::object::{Self, ID, UID};
    use sui::transfer;
    use sui::tx_context::{Self, TxContext};
    use std::option::{Self, Option};

    /// Lock is empty, nothing to take.
    const ELockIsEmpty: u64 = 0;

    /// Key does not match the Lock.
    const EKeyMismatch: u64 = 1;

    /// Lock already contains something.
    const ELockIsFull: u64 = 2;

    /// Lock that stores any content inside it.
    struct Lock<T: store + key> has key {
        id: UID,
        locked: Option<T>
    }

    /// A key that is created with a Lock; is transferable
    /// and contains all the needed information to open the Lock.
    struct Key<phantom T: store + key> has key, store {
        id: UID,
        for: ID,
    }

    /// Returns an ID of a Lock for a given Key.
    public fun key_for<T: store + key>(key: &Key<T>): ID {
        key.for
    }

    /// Lock some content inside a shared object. A Key is created and is
    /// sent to the transaction sender. For example, we could turn the
```

```
    /// lock into a treasure chest by locking some `Coin<SUI>` inside.
    ///
    /// Sender gets the `Key` to this `Lock`.
    public fun create<T: store + key>(obj: T, ctx: &mut TxContext) {
        let id = object::new(ctx);
        let for = object::uid_to_inner(&id);

        transfer::share_object(Lock<T> {
            id,
            locked: option::some(obj),
        });

        transfer::transfer(Key<T> {
            for,
            id: object::new(ctx)
        }, tx_context::sender(ctx));
    }

    /// Lock something inside a shared object using a Key. Aborts if
    /// lock is not empty or if key doesn't match the lock.
    public fun lock<T: store + key>(
        obj: T,
        lock: &mut Lock<T>,
        key: &Key<T>,
    ) {
        assert!(option::is_none(&lock.locked), ELockIsFull);
        assert!(&key.for == object::borrow_id(lock), EKeyMismatch);

        option::fill(&mut lock.locked, obj);
    }

    /// Unlock the Lock with a Key and access its contents.
    /// Can only be called if both conditions are met:
    /// - key matches the lock
    /// - lock is not empty
    public fun unlock<T: store + key>(
        lock: &mut Lock<T>,
        key: &Key<T>,
    ): T {
```

```
        assert!(option::is_some(&lock.locked), ELockIsEmpty);
        assert!(&key.for == object::borrow_id(lock), EKeyMismatch);

        option::extract(&mut lock.locked)
    }

    /// Unlock the Lock and transfer its contents to the transaction sender.
    public fun take<T: store + key>(
        lock: &mut Lock<T>,
        key: &Key<T>,
        ctx: &mut TxContext,
    ) {
        transfer::public_transfer(unlock(lock, key), tx_context::sender(ctx))
    }
}
```

Example for Sui Move is taken from the Sui Doc.