The Move Book

Whenever execution encounters an abort, transaction fails and abort code is returned to the caller. Move VM returns the module name that aborted the transaction and the abort code. This behavior is not fully transparent to the caller of the transaction, especially when a single function contains multiple calls to the same function which may abort. In this case, the caller will not know which call aborted the transaction, and it will be hard to debug the issue or provide meaningful error message to the user.

The example above illustrates the case when a single function contains multiple calls which may abort. If the caller of the do_something function receives an abort code 0 , it will be hard to understand which call to module_b::get_field aborted the transaction. To address this problem, there are common patterns that can be used to improve error handling.

It is considered a good practice to provide a safe "check" function that returns a boolean value indicating whether an operation can be performed safely. If the module_b provides a function has_field that returns a boolean value indicating whether a field exists, the do_something function can be rewritten as follows:

By adding custom checks before each call to module_b::get_field , the developer of the module_a takes control over the error handling. And it allows implementing the second rule.

The second trick, once the abort codes are handled by the caller module, is to use different abort codes for different scenarios. This way, the caller module can provide a meaningful error message to the user. The module_a can be rewritten as follows:

Now, the caller module can provide a meaningful error message to the user. If the caller receives an abort code 0 , it can be translated to "Field 1 does not exist". If the caller receives an abort code 1 , it can be translated to "Field 2 does not exist". And so on.

A developer is often tempted to add a public function that would assert all the conditions and abort the execution. However, it is a better practice to create a function that returns a boolean value instead. This way, the caller module can handle the error and provide a meaningful error message to the user.

This module can be rewritten as follows:

Utilizing these three rules will make the error handling more transparent to the caller of the transaction, and it will allow other developers to use custom abort codes in their modules.

Rule 1: Handle all possible scenarios

It is considered a good practice to provide a safe "check" function that returns a boolean value indicating whether an operation can be performed safely. If the module_b provides a function has_field that returns a boolean value indicating whether a field exists, the do_something function can be rewritten as follows:

```bash module book::module_a { use book::module_b;

const ENoField: u64 = 0;

public fun do_something() {
    assert!(module_b::has_field(1), ENoField);
    let field_1 = module_b::get_field(1);
    /* ... */
    assert!(module_b::has_field(2), ENoField);
    let field_2 = module_b::get_field(2);
    /* ... */
    assert!(module_b::has_field(3), ENoField);
    let field_3 = module_b::get_field(3);
}

} ```

By adding custom checks before each call to module_b::get_field , the developer of the module_a takes control over the error handling. And it allows implementing the second rule.

The second trick, once the abort codes are handled by the caller module, is to use different abort codes for different scenarios. This way, the caller module can provide a meaningful error message to the user. The module_a can be rewritten as follows:

```bash module book::module_a { use book::module_b;

```
const ENoFieldA: u64 = 0;
const ENoFieldB: u64 = 1;
const ENoFieldC: u64 = 2;

public fun do_something() {
    assert!(module_b::has_field(1), ENoFieldA);
    let field_1 = module_b::get_field(1);
    /* ... */
    assert!(module_b::has_field(2), ENoFieldB);
    let field_2 = module_b::get_field(2);
    /* ... */
    assert!(module_b::has_field(3), ENoFieldC);
    let field_3 = module_b::get_field(3);
}
```

} ```

Now, the caller module can provide a meaningful error message to the user. If the caller receives an abort code 0 , it can be translated to "Field 1 does not exist". If the caller receives an abort code 1 , it can be translated to "Field 2 does not exist". And so on.

A developer is often tempted to add a public function that would assert all the conditions and abort the execution. However, it is a better practice to create a function that returns a boolean value instead. This way, the caller module can handle the error and provide a meaningful error message to the user.

```bash module book::some_app_assert {

```
const ENotAuthorized: u64 = 0;

public fun do_a() {
    assert_is_authorized();
    // ...
}

public fun do_b() {
    assert_is_authorized();
    // ...
}

/// Don't do this
public fun assert_is_authorized() {
    assert!(/* some condition */ true, ENotAuthorized);
}
```

} ```

This module can be rewritten as follows:

```bash module book::some_app { const ENotAuthorized: u64 = 0;

```
public fun do_a() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun do_b() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun is_authorized(): bool {
    /* some condition */ true
}

// a private function can still be used to avoid code duplication for a case
// when the same condition with the same abort code is used in multiple places
fun assert_is_authorized() {
    assert!(is_authorized(), ENotAuthorized);
}
```

} ```

Utilizing these three rules will make the error handling more transparent to the caller of the transaction, and it will allow other

developers to use custom abort codes in their modules.

Rule 2: Abort with different codes

The second trick, once the abort codes are handled by the caller module, is to use different abort codes for different scenarios. This way, the caller module can provide a meaningful error message to the user. The module_a can be rewritten as follows:

```bash module book::module_a { use book::module_b;

```
const ENoFieldA: u64 = 0;
const ENoFieldB: u64 = 1;
const ENoFieldC: u64 = 2;

public fun do_something() {
    assert!(module_b::has_field(1), ENoFieldA);
    let field_1 = module_b::get_field(1);
    /* ... */
    assert!(module_b::has_field(2), ENoFieldB);
    let field_2 = module_b::get_field(2);
    /* ... */
    assert!(module_b::has_field(3), ENoFieldC);
    let field_3 = module_b::get_field(3);
}
```

} ```

Now, the caller module can provide a meaningful error message to the user. If the caller receives an abort code 0 , it can be translated to "Field 1 does not exist". If the caller receives an abort code 1 , it can be translated to "Field 2 does not exist". And so on.

A developer is often tempted to add a public function that would assert all the conditions and abort the execution. However, it is a better practice to create a function that returns a boolean value instead. This way, the caller module can handle the error and provide a meaningful error message to the user.

```bash module book::some_app_assert {

```
const ENotAuthorized: u64 = 0;

public fun do_a() {
    assert_is_authorized();
    // ...
}

public fun do_b() {
    assert_is_authorized();
    // ...
}

/// Don't do this
public fun assert_is_authorized() {
    assert!(/* some condition */ true, ENotAuthorized);
}
```

} ```

This module can be rewritten as follows:

```bash module book::some_app { const ENotAuthorized: u64 = 0;

```
public fun do_a() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun do_b() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun is_authorized(): bool {
    /* some condition */ true
```
```

```
}

// a private function can still be used to avoid code duplication for a case
// when the same condition with the same abort code is used in multiple places
fun assert_is_authorized() {
    assert!(is_authorized(), ENotAuthorized);
}

} ```
}
```

Utilizing these three rules will make the error handling more transparent to the caller of the transaction, and it will allow other developers to use custom abort codes in their modules.

Rule 3: Return bool instead of assert

A developer is often tempted to add a public function that would assert all the conditions and abort the execution. However, it is a better practice to create a function that returns a boolean value instead. This way, the caller module can handle the error and provide a meaningful error message to the user.

```bash module book::some_app_assert {

const ENotAuthorized: u64 = 0;

public fun do_a() {
    assert_is_authorized();
    // ...
}

public fun do_b() {
    assert_is_authorized();
    // ...
}

/// Don't do this
public fun assert_is_authorized() {
    assert!(/* some condition */ true, ENotAuthorized);
}

} ```
}
```

This module can be rewritten as follows:

```bash module book::some_app { const ENotAuthorized: u64 = 0;

public fun do_a() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun do_b() {
    assert!(is_authorized(), ENotAuthorized);
    // ...
}

public fun is_authorized(): bool {
    /* some condition */ true
}

// a private function can still be used to avoid code duplication for a case
// when the same condition with the same abort code is used in multiple places
fun assert_is_authorized() {
    assert!(is_authorized(), ENotAuthorized);
}

} ```
}
```

Utilizing these three rules will make the error handling more transparent to the caller of the transaction, and it will allow other developers to use custom abort codes in their modules.