The Move Book

To talk about best practices for upgradeability, we need to first understand what can be upgraded in a package. The base premise of upgradeability is that an upgrade should not break public compatibility with the previous version. The parts of the module which can be used in dependent packages should not change their static signature. This applies to modules - a module can not be removed from a package, public structs - they can be used in function signatures and public functions - they can be called from other packages.

To discard previous versions of the package, the objects can be versioned. As long as the object contains a version field, and the code which uses the object expects and asserts a specific version, the code can be force-migrated to the new version. Normally, after an upgrade, admin functions can be used to update the version of the shared state, so that the new version of code can be used, and the old version aborts with a version mismatch.

There's a common pattern in Sui which allows changing the stored configuration of an object while retaining the same object signature. This is done by keeping the base object simple and versioned and adding an actual configuration object as a dynamic field. Using this anchor pattern, the configuration can be changed with package upgrades while keeping the same base object signature.

This section is coming soon!

Versioning objects

To discard previous versions of the package, the objects can be versioned. As long as the object contains a version field, and the code which uses the object expects and asserts a specific version, the code can be force-migrated to the new version. Normally, after an upgrade, admin functions can be used to update the version of the shared state, so that the new version of code can be used, and the old version aborts with a version mismatch.

```bash module book::versioned_state {

~~~
const EVersionMismatch: u64 = 0;

const VERSION: u8 = 1;

/// The shared state (can be owned too)
public struct SharedState has key {
    id: UID,
    version: u8,
    /* ... */
}

public fun mutate(state: &mut SharedState) {
    assert!(state.version == VERSION, EVersionMismatch);
    // ...
}
~~~

} ```

There's a common pattern in Sui which allows changing the stored configuration of an object while retaining the same object signature. This is done by keeping the base object simple and versioned and adding an actual configuration object as a dynamic field. Using this anchor pattern, the configuration can be changed with package upgrades while keeping the same base object signature.

```bash module book::versioned_config { use sui::vec_map::VecMap; use std::string::String;

~~~
/// The base object
public struct Config has key {
    id: UID,
    version: u16
}

/// The actual configuration
public struct ConfigV1 has store {
    data: Bag,
    metadata: VecMap<String, String>
}

// ...
~~~

} ```

This section is coming soon!

## Versioning configuration with dynamic fields

There's a common pattern in Sui which allows changing the stored configuration of an object while retaining the same object signature. This is done by keeping the base object simple and versioned and adding an actual configuration object as a dynamic field. Using this anchor pattern, the configuration can be changed with package upgrades while keeping the same base object signature.

```bash module book::versioned_config { use sui::vec_map::VecMap; use std::string::String;

~~~
/// The base object
public struct Config has key {
    id: UID,
    version: u16
}

/// The actual configuration
public struct ConfigV1 has store {
    data: Bag,
    metadata: VecMap<String, String>
}

// ...
~~~

} ```

This section is coming soon!

## Modular architecture

This section is coming soon!
```