The Move Book

Sometimes, there's a need to create a new type that behaves similarly to an existing type but with certain modifications or restrictions. For example, you might want to create a collection type that behaves like a vector but doesn't allow modifying the elements after they've been inserted. The wrapper type pattern is an effective way to achieve this.

The wrapper type pattern is a design pattern in which you create a new type that wraps an existing type. The wrapper type is distinct from the original but can be converted to and from it.

Often, it is implemented as a positional struct with a single field.

In cases where the goal is to extend the behavior of an existing type, it is common to provide accessors for the wrapped type. This approach allows users to access the underlying type directly when needed. For example, in the following code, we provide the inner() , inner_mut() , and into_inner() methods for the Stack type.

The wrapper type pattern offers several benefits:

The wrapper type pattern is powerful in two scenarios—when you want to limit the behavior of an existing type while providing a custom interface to the same data structure, and when you want to extend the behavior of an existing type. However, it does have some limitations:

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to store large collections of data in a more flexible, albeit more expensive, way.

Definition

The wrapper type pattern is a design pattern in which you create a new type that wraps an existing type. The wrapper type is distinct from the original but can be converted to and from it.

Often, it is implemented as a positional struct with a single field.

```bash module book::wrapper_type_pattern;

/// Very simple stack implementation using the wrapper type pattern. Does not allow /// accessing the elements unless they are popped. public struct Stack(vector) has copy, store, drop;

/// Create a new instance by wrapping the value. public fun new(value: vector): Stack { Stack(value) }

/// Push an element to the stack. public fun push_back(v: &mut Stack, el: T) { v.0.push_back(el); }

/// Pop an element from the stack. Unlike `vector`, this function won't /// fail if the stack is empty and will return `None` instead. public fun pop_back(v: &mut Stack): Option { if (v.0.length() == 0) option::none() else option::some(v.0.pop_back()) }

/// Get the size of the stack. public fun size(v: &Stack): u64 { v.0.length() } ```

In cases where the goal is to extend the behavior of an existing type, it is common to provide accessors for the wrapped type. This approach allows users to access the underlying type directly when needed. For example, in the following code, we provide the inner() , inner_mut() , and into_inner() methods for the Stack type.

```bash /// Allows reading the contents of theStack`. public fun inner(v: &Stack): &vector { &v.0 }

/// Allows mutable access to the contents of the `Stack`. public fun inner_mut(v: &mut Stack): &mut vector { &mut v.0 }

/// Unpacks the `Stack` into the underlying `vector`. public fun into_inner(v: Stack): vector { let Stack(inner) = v; inner } ```

The wrapper type pattern offers several benefits:

The wrapper type pattern is powerful in two scenarios—when you want to limit the behavior of an existing type while providing a custom interface to the same data structure, and when you want to extend the behavior of an existing type. However, it does have some limitations:

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to

store large collections of data in a more flexible, albeit more expensive, way.

## Common Practices

In cases where the goal is to extend the behavior of an existing type, it is common to provide accessors for the wrapped type. This approach allows users to access the underlying type directly when needed. For example, in the following code, we provide the inner() , inner_mut() , and into_inner() methods for the Stack type.

``bash /// Allows reading the contents of the`Stack`. public fun inner(v: &Stack): &vector { &v.0 }

/// Allows mutable access to the contents of the `Stack`. public fun inner_mut(v: &mut Stack): &mut vector { &mut v.0 }

/// Unpacks the `Stack` into the underlying `vector`. public fun into_inner(v: Stack): vector { let Stack(inner) = v; inner } ```

The wrapper type pattern offers several benefits:

The wrapper type pattern is powerful in two scenarios—when you want to limit the behavior of an existing type while providing a custom interface to the same data structure, and when you want to extend the behavior of an existing type. However, it does have some limitations:

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to store large collections of data in a more flexible, albeit more expensive, way.

## Advantages

The wrapper type pattern offers several benefits:

The wrapper type pattern is powerful in two scenarios—when you want to limit the behavior of an existing type while providing a custom interface to the same data structure, and when you want to extend the behavior of an existing type. However, it does have some limitations:

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to store large collections of data in a more flexible, albeit more expensive, way.

## Disadvantages

The wrapper type pattern is powerful in two scenarios—when you want to limit the behavior of an existing type while providing a custom interface to the same data structure, and when you want to extend the behavior of an existing type. However, it does have some limitations:

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to store large collections of data in a more flexible, albeit more expensive, way.

## Next Steps

The wrapper type pattern is very useful, particularly when used in conjunction with collection types, as demonstrated in the previous section. In the next section, we will cover Dynamic Fields — an important primitive that enables Dynamic Collections , a way to store large collections of data in a more flexible, albeit more expensive, way.