Don’t Go Changing: How to Code Immutable UVM Objects

William L. Moore
WilliamL33Moore@gmail.com

Abstract – In object-oriented programming (OOP), immutable objects enhance the simplicity, reliability, and performance of software. This paper introduces immutable objects, explains how to create them in SystemVerilog, and addresses challenges in incorporating them within the Universal Verification Methodology (UVM).

Introduction

Design verification (DV) engineers develop code for testbenches that simulate integrated circuits (ICs). High-quality testbench code is critical for reliable and thorough verification of an IC project. Like any software, testbenches must satisfy requirements, have no defects, and be maintainable. Commercial projects are invariably time-sensitive, so DV engineers also strive to write code efficiently, taking advantage of best practices for architecture, design, implementation, and collaboration. To realize these goals, DV engineers can adopt techniques that arose to improve enterprise software development. In particular, Domain-Driven Design (DDD) propounds strategies and patterns for creating accurate and appropriate software models of application domain elements. The concept of *immutable value objects* is a fundamental building block of DDD. For the DV engineer, the device under test (DUT) is the domain. The engineer can improve testbench code by identifying its domain-specific value objects and making them immutable. The SystemVerilog language readily supports immutable classes, but some aspects of their creation are non-obvious. UVM base classes and creational patterns are less conducive to immutability. This paper offers best practices for implementing immutable objects in SystemVerilog, proposals for overcoming the challenges of UVM, and illustrative examples from the domain of IC verification methodology.

Immutable Objects in OOP

An immutable object is a value object whose values are initialized atomically when the object is created, and whose public interfaces always return the same values throughout the lifetime of the object. In short, it’s a value object whose values never change. A value object is an object defined by its values, not its identity; if two value objects have the same values, then the value objects are equal. In contrast to immutable objects, mutable objects have public mutating methods, and their values can change over time. Many OOP languages provide built-in or standard libraries of primitive immutable classes such as numbers, strings, and tuples. This paper is more concerned with higher-level domain-specific value objects, often recursively composed of primitives and other objects. In this paper, “developers” refers to the creators of the immutable class, and “users” signifies those who use the class in their “user code.” User visibility into the class is limited to the public interfaces exposed by the developers.

Immutable objects simplify user code since they are effectively constants. Processes can share an immutable object by reference without the risk of its contents changing, avoiding the cost of passing by value. In some applications, immutable objects can save memory and improve efficiency because it is unnecessary to make an immutable copy of an immutable object; just keep using the same instance. In contrast, shared mutable objects are a well-known source of defects in OOP. If multiple processes each have a reference to a shared mutable object, there is a risk that one could intentionally or inadvertently change a value, with unwanted consequences for subsequent processes. One standard remedy is so-called defensive copying, where user code makes a copy of received mutable objects to ensure a stable snapshot. Such copies incur memory, performance, and complexity costs. Immutable objects improve communication and increase understanding among software developers, testers, and users because such objects have simpler interfaces. Immutable objects have no mutating “setter” methods or side effects to document, and their use cases after initialization often reduce to simply “getting” values. Immutable objects model real-life immutable domain concepts well, such as geographical coordinates, date ranges, and revision control commits. Immutable objects enhance code cohesion and clarity. For starters, immutability creates a demarcation between an object’s sole creator and any number of consumers. Only the creator can set the object’s values for all the consumers to consume. Once it is determined that a class should be immutable, the developer may purge it of mutable fields and methods, which results in greater cohesion. These benefits potentially contribute to cleaner user code with fewer defects.

There are disadvantages. It takes developer effort to ensure that the object is truly immutable—to rid the class of mutating “leaks” rigorously. There is no standard automated test for immutability. Developers can only apply their best effort to create a class that is immutable by design, and convey that declaration to users via documentation. It may be inevitable that a developer or user needs to make a mutable copy of an immutable object or vice versa, and both operations are costly in terms of code, memory, and performance, compared to operating on a mutable object in place. This paper demonstrates that introducing immutable objects into systems that weren't designed for them requires significant rework. Legacy code that freely modifies mutable objects must change when the objects become immutable. It is easier to design immutability into systems from the beginning.

Developers can mimic immutable behavior with mutable objects whose mutating methods are disabled depending on the state of the object or environment at run-time. For example, UVM's `uvm_get_to_lock_dap` data access policy allows the user to set a value any number of times, but then locks the value once it is retrieved via "get," such that any subsequent attempt to "set" results in an error. This paper strives for true immutable objects that present no public mutating methods. Some objects may be considered hybrid objects, containing both immutable and mutable fields. This paper includes examples of such objects in use in UVM.

Immutability is not the same as persistence. User code can create and destroy immutable objects freely.

Immutable Objects in SystemVerilog

Use these guidelines when designing classes for immutable objects in SystemVerilog. It's up to the developer's discretion which guidelines to follow, but omitting certain ones opens the class up to possible mutation.

1. Declare all variables as `local`.
2. Provide public "getter" functions for private variables.
3. Do not provide public or protected "setters."
4. Provide the constructor `new()` enough parameters to initialize every variable.
5. Qualify `new()` as `local`.
6. Provide a static factory method.
7. Do not modify variables after the constructor finishes.
8. Check the entire chain of parent classes for inherited mutable members and mutating methods.
9. Do not retain or share references to mutable objects.
10. Do not qualify any variables as random.

Guideline details follow.

Declare all variables as `local`.

Public variables can be mutated freely by any user code that has a reference to the object. Protected variables are safe from outside mutation, but subclasses can mutate the superclass' protected variables, which means that user code can have what appears to be a reference to an immutable superclass object that actually refers to a mutable subclass object. Therefore, the safest solution is to declare all variables as private with the SystemVerilog `local` keyword.

Variables can also be declared as instance constants with the `const` keyword. This is not recommended because the syntax is quite restrictive; instance constants can only be assigned once, inside the constructor.

Provide public "getter" functions for private variables.

Private variables are not visible to user code, so provide public "getter" functions that directly or indirectly return any desired private values. Not all local variables need to be exposed publicly.

Do not provide public or protected "setters."

A "setter" is a function that changes the value of a class variable, so immutable classes must not have any public or protected setters that user code or subclasses could use to mutate an object. An immutable class may safely have private `local` setters solely for the purpose of initializing values in the constructor.

In OOP, it is common practice for mutable classes to have private variables and public getters and setters. In contrast, immutable classes have private variables and public getters, but setters are omitted or kept private.

Provide the constructor `new()` enough parameters to initialize every variable.

Immutable objects are initialized atomically when they are created. The natural way to initialize values is to pass them as arguments to the constructor, `new()`. Often there is a one-to-one relationship between local class variables and the constructor parameters that initialize them, but other cardinal relationships are possible. For example, multiple variables could be initialized by members of a single aggregate parameter, e.g., an array, struct, or object. Conversely, one variable could be initialized by a function of multiple parameters, such as a calculation or concatenation. Regardless, the object must be completely initialized by the constructor, so the constructor must receive enough data through its arguments, or other means, to do so. (A subsequent section of this paper discusses alternative strategies for the constructor to receive initializing values.)

Qualify `new()` as `local`.

As explained earlier, a user subclass of an immutable superclass can mutate protected variables and add new mutable variables, resulting in a mutable object. A drastic measure to prevent this is for the developer to disallow extension.

SystemVerilog provides an indirect way to block class extension: qualify the constructor `new()` as private with the `local` keyword. Subclasses must be able to call `super.new()` explicitly or implicitly. If the superclass constructor is `local`, any subclass will result in a compilation error.

IEEE 1800-2023 introduces a direct way to block class extension: the class `:final` specifier, including the colon. Ref. [2] states: "An attempt to extend a class that was specified as final shall result in an error." The `:final` specifier can also be applied to individual methods. Older simulator versions may not support the `:final` specifier, so it is more compatible and portable to make the constructor `local`.

Provide a static factory method.

If the constructor is private, then the class must have at least one static factory method. The simplest approach is for the factory method to have the same parameters as the constructor, to pass its arguments along to the constructor, and to return the result. This paper consistently uses the name `create_new()` for such a pass-through factory function. The class may have additional factory functions, perhaps with different parameters, such as a static copy creator that takes a single object parameter as the "source." Or, different factory functions may return different types. For example, one function may return a concrete object handle, and another may return an abstract superclass handle.

Do not modify variables after the constructor finishes.

All variables must be initialized by the time the constructor finishes. The developer must ensure by design and inspection that no public or private class methods have the direct or indirect side effect of mutating any immutable variables. If the class has private setter methods, a prudent rule is that they may only be called by the constructor.

Check the entire chain of parent classes for inherited mutable members and mutating methods.

If an immutable class extends no superclasses and has no subclasses, then the developer has total control over the class definition and its instances. However, if there is a chain of one or more superclasses, and any of them have mutable variables or mutating methods, then the developer's class inherits them all and becomes mutable. Therefore, the developer should review all parent classes and identify any mutable variables and mutating methods.

If the developer owns a superclass, they have the option of modifying it to remove vulnerabilities. Otherwise, if the developer is extending a superclass that they cannot modify, they can take steps to mitigate unwanted mutable incursions. Chiefly, a subclass can override a public variable with a compatible private variable with the same name. Likewise, a subclass can override a mutating method with a compatible method with the same signature that does not perform the mutation. The overriding method in the subclass should not call the respective superclass method via `super`. The developer can decide the severity of an attempt to call a forbidden setter and respond accordingly, from quietly ignoring the attempt, to issuing an info, warning, error, or fatal message. Overriding variables and methods is not a perfect solution because user code can always upcast the object and get access to the mutable variables and mutating methods through the casted superclass reference. However, if a mutating method in the superclass happens to be virtual, then the immutable subclass' override of the method always takes precedence, and user code can no longer access the mutating version, making the subclass more immutable.

Do not retain or share references to mutable objects.

Consider a class that has a private class variable for holding a reference to a sub-object. The developer intends for the class to be immutable, so the class initializes the private variable through a constructor argument, and provides a simple getter that shares a reference to the sub-object with user code. If the user initializes the variable with a reference to a known immutable sub-object, then there is no issue. The class' getter always returns a reference to the same immutable object, which never changes, and the class' immutability is preserved. However, if the user initializes the class variable with a reference to a mutable sub-object, then the developer's class is no longer immutable; the variable presents a mutable breach. User code can use the getter to get a reference, which can then be used to mutate the sub-object.

At a high level, the remedy is that an immutable class should not retain or share references to mutable objects. One common approach is for the immutable class to make a defensive private deep copy of any received objects, and to ensure that its own private copy never changes. Likewise, every time the getter is called, it must make a fresh deep copy of the internal mutable object, and return a reference to that, so that its internal object is never shared or exposed.

Do not qualify any variables as random.

Every SystemVerilog class has a built-in `randomize()` function, with which the user can mutate any object. The developer can prevent randomization syntactically by not declaring any variables as random (keyword `rand`); then `randomize()` has no effect. There are run-time alternatives for disabling randomization, such as customizing `pre_randomize()` or the random constraints.

Randomization is a necessary feature of most SystemVerilog testbenches and applications, so in a later section, this paper addresses ways to support both randomization and immutability.

Immutable Objects in UVM

UVM provides SystemVerilog base classes for integrated circuit verification. Unfortunately, they are not immutable, not even `uvm_object`, and UVM's creational patterns are not conducive to immutability. This section incorporates immutable objects in UVM and offers solutions to the challenges. Two use cases from the UVM User's Guide are examined: configuration objects and sequence items.

An Immutable Configuration Object

Testbench components should be configurable, so UVM recommends configuration classes—`uvm_object`-based value objects that encapsulate random build phase parameters. Configuration objects, especially shared ones, shouldn't change, so make them immutable. Configuration objects are initialized once in the build phase, then user code can access their values freely without the possibility of them changing intentionally or inadvertently.

This discussion is presented as a tutorial, in which we imagine a component class called `box` that is configured with integer knobs for length, width, and height. As the developer, you will implement an immutable configuration class called `box_config` that encapsulates the three integer knobs.

First, create an abstract `virtual` base class that extends `uvm_object`, called `immutable_object`. This need only be done once per project; `immutable_object` can serve as a base class for numerous immutable objects. Override these virtual mutating methods inherited from `uvm_object`:

- `set_name()`
- `do_copy()`
- `do_unpack()`
- `set_int_local()`
- `set_string_local()`
- `set_object_local()`

Also, override the built-in function `pre_randomize()`. The override implementations must not perform the indicated mutations, nor call the mutating superclass versions via `super`, and they should throw a `` `uvm_fatal() `` error so that if user code tries to call the methods, they fail visibly at runtime, inviting user analysis. Individual subclasses have the option of overriding these virtual methods if they require a severity lower than fatal. Note that extending `immutable_object` does not guarantee that a class is immutable; nothing prevents a subclass from containing mutable variables and mutating methods. The `immutable_object` class is for convenience and reuse, and the name communicates the intent that any subclasses *should* be immutable.

Code `box_config` (Fig. 1), following the SystemVerilog guidelines from the earlier section. In addition, follow these additional steps specific to UVM or to our `box` domain application. These steps are illustrative but apply generally to the creation of any prospective immutable class. As with the SystemVerilog guidelines, some of these steps are syntactic imperatives, and some are optional at the developer's discretion. Omitting some optional steps risks opening the class up to mutation.

```
typedef class box_config_immutable;
typedef class box_config_factory_generic;


// Class extends abstract base class box_config_immutable, which extends
// abstract base class immutable_object and implements box_config_interface
// ========================================================================

class box_config extends box_config_immutable;

    // * Typedef "factory_type" associates this class with its own factory class
    typedef box_config_factory_generic#(box_config) factory_type;

    // * Local variables
    // * No const variables
    // * No rand variables
    local int length;
    local int width;
    local int height;

    // * No `uvm_object_utils()
    // * Field utils only
    // * Set field flags "UVM_NOPACK | UVM_NOCOPY | UVM_READONLY" on every variable
    `uvm_field_utils_begin(box_config)
        `uvm_field_int(length, UVM_ALL_ON | UVM_NOPACK | UVM_NOCOPY | UVM_READONLY);
        `uvm_field_int(width , UVM_ALL_ON | UVM_NOPACK | UVM_NOCOPY | UVM_READONLY);
        `uvm_field_int(height, UVM_ALL_ON | UVM_NOPACK | UVM_NOCOPY | UVM_READONLY);
    `uvm_field_utils_end

    // * Local constructor "new()"
    // * First parameter is "string name"
    // * Remaining parameters initialize all variables
    // * All parameters have default values
    local function new (string name="", int length=0, int width=0, int height=0);
        super.new(name);
        this.set_length(length);
        this.set_width(width);
        this.set_height(height);
    endfunction

    // * Static factory method
    static function box_config_immutable create_new (
            string name="", int length=0, int width=0, int height=0
        );
        box_config product = new(name, length, width, height);
        return product;
    endfunction

    // * Static copy function
    static function box_config_immutable create_copy (string name="", uvm_object rhs);
        // Non-trivial copy algorithm is provided by a parameterized helper class
        create_copy = box_config_copier#(box_config)::create_copy(name, rhs);
    endfunction

    //* Required implementation of uvm_object::get_type_name()
    virtual function string get_type_name ();
        return "box_config";
    endfunction

    // * Required implementation of uvm_object::create()
    virtual function uvm_object create (string name="");
        box_config object = new(name);
        return object;
    endfunction

    // Public getters required by box_config_interface
    // ------------------------------------------------
    virtual function int get_length ();
        return this.length;
    endfunction
    virtual function int get_width ();
        return this.width;
    endfunction

    virtual function int get_height ();
        return this.height;
    endfunction

    // Local setters
    // -------------
    local function void set_length (int length);
        this.length = length;
    endfunction

    local function void set_width (int width);
        this.width = width;
    endfunction

    local function void set_height (int height);
        this.height = height;
    endfunction
endclass
```

Figure 1. Source code for immutable `box_config` class.

1. Create an interface class, `box_config_interface`, that defines your pure virtual getter prototypes.
2. Create a virtual base class, `box_config_immutable`.
3. Declare immutable class `box_config`, extending `box_config_immutable`.
4. Add `local` variables `length`, `width`, and `height`.
5. Use `` `uvm_field_utils_* `` macros instead of `` `uvm_object_utils_* ``.
6. Add `` `uvm_field_*() `` macros for the variables and set flags that support immutability.
7. Declare constructor `new()` with parameters for `name` and all initializing values.
8. Implement a static factory function and a static copy function.
9. Override required `uvm_object` methods `create()` and `get_type_name()` explicitly.
10. Implement the public getters mandated by `box_config_interface`.

Step details follow.

Create an interface class, `box_config_interface`, that defines your pure virtual getter prototypes.

The interface class defines pure virtual prototypes for `box_config`:

- `pure virtual function int get_length();`
- `pure virtual function int get_width();`
- `pure virtual function int get_height();`

We omit protoypes for setters, so this represents the simplified read-only interface of an immutable object. This interface class is optional, but it lays the foundation for a family of compatible mutable and immutable classes that share it.

Create a virtual base class, `box_config_immutable`.

This is an abstract base class that extends `immutable_object` and implements `box_config_interface`. It is optional, but it extends the foundation to a family of specifically immutable classes.

Define default implementations for your static factory functions `create_new()` and `create_copy()`, described below. The default implementations can be placeholders that throw `` `uvm_fatal() `` messages, which forces concrete subclasses to provide their own implementations.

Also, define default implementations for the pure virtual prototypes from `box_config_interface`. It is sufficient to copy the pure virtual prototypes into the abstract base class, leaving actual implementations to the concrete subclasses.

Declare immutable class `box_config`*, extending* `box_config_immutable`*.*

Finally we can begin coding our target `box_config` class. It extends base class `box_config_immutable`, which means it inherits members from `uvm_object`, immutability-enforcing implementations from `immutable_object`, and box behavior prototypes from `box_config_interface`. The remaining steps in this section build out `box_config` incrementally. Fig. 1 contains the complete code listing for the class.

Add `local` *variables* `length`*,* `width`*, and* `height`*.*

Following the guidelines, add private `local` integer variables `length`, `width`, and `height`. Do not use the `const` keyword because it is incompatible with the UVM field utility macros and would result in compilation failure. The macros assign to field variables outside the constructor; this is illegal for `const` variables.

Use `` `uvm_field_utils_* `` *macros instead of* `` `uvm_object_utils_* ``*.*

Because our immutable class will have a private constructor, you should not use the `` `uvm_object_utils_* `` macros. They make use of constructors without initializing arguments and the mutating functions that we overrode with fatal implementations, so the macros are unfortunately unsuitable for use in immutable classes. They rely on a creational pattern where objects are created and initialized in separate steps, which is antithetical to the requirement that immutable objects be created and initialized atomically.

Fortunately, UVM provides suitable alternative macros: use `` `uvm_field_utils_begin() `` and `` `uvm_field_utils_end `` instead. They expand to a subset of the code of `` `uvm_object_utils_* `` without the parts problematic to immutability.

Without `` `uvm_object_utils_* ``, your class is not registered with the UVM factory. An upcoming section details how to get the creational benefits of the factory even without direct registration.

Add `` `uvm_field_*() `` *macros for the variables and set flags that support immutability.*

Between `` `uvm_field_utils_begin() `` and `` `uvm_field_utils_end ``, add `` `uvm_field_*() `` macros for the three value fields. In addition to the typical field flags, e.g., `UVM_ALL_ON`, set the following flags with a bitwise OR. For UVM 1.2 and earlier, set flags `UVM_NO_PACK | UVM_NOCOPY | UVM_READONLY`. For UVM 1800-2017 and later, set `UVM_NO_PACK | UVM_NOCOPY | UVM_NOSET`. These are runtime flags that bypass the mutating `do_unpack()`, `do_copy()`, and `set_*_local()` methods we overrode and disabled in the abstract `immutable_object` base class. These flags are optional and redundant, erring on the side of caution and clarity. The UVM bypasses issue warnings, which are lower severity than the fatal messages in our overrides.

Declare constructor `new()` *with parameters for* `name` *and all initializing values.*

Following the guidelines, declare a private `local` constructor, `new()`. Your constructor must be compatible with the `uvm_object` base class `new()` function, so the first argument must be `string name`, and all remaining arguments must have default values, so that a call to `box_config::new("some_string")`, with only one argument, would compile without "missing argument" errors.

The remaining arguments are our three initializing integer values, `length`, `width`, and `height`, all with reasonable default values, zero. We call private setters from the constructor to initialize the three class variables.

Implement a static factory function and a static copy function.

We adhere to our convention of naming the static factory function `create_new()`, and give it the same arguments as the constructor for easy pass-through. Since we are ultimately building a family of box configuration classes, our factory function returns an abstract base `box_config_immutable` handle, rather than a concrete `box_config` handle. This is a design decision for the developer to make, depending on the application. The developer could even implement both abstract and concrete static factory functions in the same class.

Next we write a static copy function. This is optional. A user might reasonably want to create an immutable copy of a mutable object. We cannot use `uvm_object::copy()` on our immutable object—we disabled it—so instead, we offer a static copy function called `create_copy()` that takes `string name` and `uvm_object rhs` arguments, just like `uvm_object::copy()`. Our implementation is not shown here, but we delegate to a parameterized copy class that casts `rhs` to a `box_config_interface` reference, then uses the interface class getters to initialize a new `box_config` object, again opting to return an abstract base `box_config_immutable` handle instead of a concrete one. Ref. [15] hosts the complete source code for this example.

We have not implemented a custom `uvm_object::clone()` function since there is generally no need to clone an immutable object; one copy can serve all users. An attempt to clone a `box_config` object with the default implementation would try to call our `immutable_object::do_copy()` override and would trigger its fatal message. A developer may choose to implement `clone()` explicitly for parity with mutable UVM objects. The simplest implementation is a virtual function wrapper around the static copy function.

```
virtual function uvm_object clone ();
    return box_config::create_copy(this.get_name(), this);
endfunction
```

Override required `uvm_object` methods `create()` and `get_type_name()` explicitly.

The `` `uvm_object_utils_* `` macros automatically implement required `uvm_object` virtual functions `create()` and `get_type_name()`. We do not use `` `uvm_object_utils_* ``, so we must implement the two mandatory functions ourselves. The `create()` function takes no initializing arguments, so it always returns an immutable object with default values, in our case all zeroes, so it is not practically useful.

Implement the public getters mandated by `box_config_interface`.

Finally, implement the three getters for our class variables to satisfy the prototypes in `box_config_interface`. We also choose to implement three private `local` setters.

Our immutable `box_config` class is ready for use. User code creates an instance with the factory function:

```
int length = 24;
int width = 18;
int height = 12;
box_config_immutable box_cfg = box_config::create_new("box_cfg", length, width, height);
```

A TALE OF TWO FACTORIES

The UVM factory allows the user to choose at runtime which UVM object or component to create. Furthermore, the override facility can replace the hard-coded target class with a compatible subclass. Since you can't register your immutable class with the UVM factory, you can't do runtime overrides. Here is a proposed solution: for each immutable class, create a small secondary factory class whose sole function is to create that specific type of immutable object, and which is registered with the UVM factory. The secondary factory is effectively a UVM factory-friendly creational proxy for the immutable class. The user creates an immutable object in two steps. First they use the UVM factory to instantiate a secondary factory object, then use that secondary factory to create the target immutable product. To create an immutable object of a different type, they override the secondary factory type with a different factory type that is a subclass of the original factory, and that creates the wanted immutable object.

Our example started with an immutable `box_config` class with length, width, and height. We now expand that to a family of immutable classes by also creating two variants: `box_config_variant_cube`, a cube whose length, width, and height are always the same, and `box_config_variant_rectangle`, a flat object whose height is always zero. All three variants extend `box_config_immutable`, so they are compatible with each other.

Fig. 2 has a complete listing for `box_config_factory`, a secondary factory class that has virtual functions for creating immutable `box_config` objects. The functions are `create_new()` and `create_copy()`. They delegate to `box_config`'s respective static functions, and return handles of abstract type `box_config_immutable`. User code invokes the UVM factory to create a `box_config_factory` instance, which in turn creates a `box_config` object:

```
box_config_factory factory = box_config_factory::type_id::create("factory");
box_config_immutable box_cfg = factory.create_new("box_cfg", length, width, height);
```

We have retained a handle to the factory object so we can use it to create multiple `box_cfg` objects. Alternatively, if the user doesn't need to retain the factory handle, they can chain the two function calls together in one line. The created factory object is anonymous and is not retained.

```
box_cfg = box_config_factory::type_id::create("anonymous_factory").create_new("box_cfg", length,
    width, height);
```

```
typedef class box_config_immutable;
typedef class box_config;


// Concrete factory base class produces flagship box_config objects
// ================================================================

class box_config_factory extends uvm_object;

    // Register with the UVM factory
    `uvm_object_utils(box_config_factory)

    function new (string name="box_config_factory");
        super.new(name);
    endfunction

    // Virtual factory functions delegate to box_config static factory functions
    // -----------------------------------------------------------------------
    virtual function box_config_immutable create_new (
            string name="", int length=0, int width=0, int height=0
        );
        create_new = box_config::create_new(name, length, width, height);
    endfunction

    virtual function box_config_immutable create_copy (
            string name="", uvm_object rhs
        );
        create_copy = box_config::create_copy(name, rhs);
    endfunction
endclass


// Parameterized factory subclass produces any box_config_immutable object
// =======================================================================

class box_config_factory_generic#(type PT=box_config) extends box_config_factory;

    // Register with the UVM factory
    `uvm_object_param_utils(box_config_factory_generic#(PT))

    function new (string name="box_config_factory_generic");
        super.new(name);
    endfunction

    // Virtual factory functions delegate to product type static factory functions
    // -------------------------------------------------------------------------
    virtual function box_config_immutable create_new (
            string name="", int length=0, int width=0, int height=0
        );
        create_new = PT::create_new(name, length, width, height);
    endfunction

    virtual function box_config_immutable create_copy (
            string name="", uvm_object rhs
        );
        create_copy = PT::create_copy(name, rhs);
    endfunction
endclass
```
Figure 2. Source code for `box_config` factory classes.

We want to take advantage of factory overrides so that we can opt to create cubes and rectangles as well as standard boxes. We create a second secondary factory class, a subclass of `box_config_factory` called `box_config_factory_generic`, parameterized with type parameter `PT`, for "product type". Fig. 2 has the complete listing for this new factory class. The parameterized factory class can create all three `box_config` variants, selected by assigning the `PT` parameter. The user calls `uvm_factory::set_type_override_by_type()` to instruct the factory to override the standard box config factory with a specialized cube config factory. Now user code that requests a box config via a box config factory produces a desired cube config instead.

```
uvm_factory::get().set_type_override_by_type(box_config_factory::get_type(),
    box_config_factory_generic#(box_config_variant_cube)::get_type());
box_cfg = box_config_factory::type_id::create("anonymous_factory").create_new("box_cfg", length,
    width, height); // Same operation now creates a box_config_variant_cube
```

A `factory_type` typedef in the box config classes serves as a convenient alias for the specialized generic factory.

Half-baked Alternative Constructor Knob Strategies[1]

The secondary factories are required because the default UVM factory cannot pass values to the class constructor. The UVM factory simply calls `new(name)` with no additional arguments, so the immutable object is never initialized. Here are novel alternative ways to initialize an object atomically at creation time. They are unusual and have limitations that make them impractical and inconvenient for the user to varying degrees, but they all have the advantage that they work with the UVM factory, so no additional factory classes are required. The developer must relax some of the SystemVerilog immutability guidelines. First and foremost, all these constructors must be public. These somewhat tongue-in-cheek methods are included here as a creative sidebar to highlight the lengths required to work around the limitations of UVM's object creation implementation.

Pass values to the constructor through global storage.

User code stores initializing values in a global storage location such as the UVM resource database, UVM global pools, UVM global queues, UVM component tree, a module, or even a file. The immutable class constructor retrieves the values from the global storage.

This method is potentially inconvenient for users, and poses maintenance and portability challenges since the constructor and user code must agree on the location of the storage. This method carries the risk that user values could be corrupted before the constructor receives them, and that values could inadvertently leak between objects, e.g., one object receives initializing values left in storage from a previous object. Otherwise, this approach is relatively reasonable and in line with typical usage of global facilities in UVM and SystemVerilog.

Pass values to the constructor through static variables.

User code stores initializing values in static variables inside the class itself. This approach is more cohesive than external global storage since the variables are encapsulated inside the class definition. However, both approaches share the same risks of data corruption and leaks.

Encode simple values in the `name` argument as a whitespace-separated string.

User code stores the object's name and initializing values in a string formatted as whitespace-separated tokens, and passes that compound string to the constructor through the `name` argument. The constructor extracts the values with SystemVerilog's `$sscanf()` function.

```
// User code
box_config_immutable box_cfg = box_config::type_id::create($sformatf("%s %d %d %d", "box_cfg",
    length, width, height));

// Constructor
function new(string name="");
    string actual_name;
    int length, width, height;
    super.new("");
    void'($sscanf(name, "%s %d %d %d", actual_name, length, width, height));
    set_name(actual_name); // This immutable object must allow name changes
    set_length(length);
    set_width(width);
    set_height(height);
endfunction
```

This method works best when all the class values are simple scalar integers or single-word strings. Real numbers might lose precision, string values cannot contain whitespace, and complex values like structs, arrays, and objects would be difficult to encode and decode.

[1] H.A.C.K.S.

Use `uvm_packer` to pack complex values into the `name` argument as a non-printable string of bytes.

The `uvm_packer` policy object can serialize arbitrarily complex UVM objects into an array of bytes. User code packs initializing values into such a byte array, encodes the byte array into a non-printable string, and passes the string to the constructor via the `name` argument. The constructor decodes and unpacks the string to retrieve the values.

SystemVerilog strings may not contain the null byte 8'h00, so a lossless encoding scheme is required to ensure that the string derived from the byte array is free of zeros.

This method is baroque to the point of absurdity, but it is functional.

Pass constant values as class parameters.

The immutable class is parameterized, and user code passes initializing values as class parameters.

```
box_config_immutable box_cfg = box_config#(
    .length (24),
    .width (18),
    .height (12))::type_id::create("box_cfg");
```

A significant limitation is that class parameters must be constants. User code cannot pass variables this way.

Pass values through the `uvm_component` constructor `parent` argument.

The default `uvm_object` constructor takes only one argument, `string name`. In contrast, the default `uvm_component` constructor takes two arguments: `string name` and `uvm_component parent`. If the immutable class extends `uvm_component` instead of `uvm_object`, user code can encapsulate initializing values in a `uvm_component` object specially devised for that purpose, and pass it to the constructor through the `parent` argument. The constructor retrieves the values from `parent`.

A limitation is that objects derived from `uvm_component` cannot truly be immutable; for starters, user code can always mutate a component by adding children to it. Also, we have removed the need for a secondary factory class, but added the need for a special component class for `parent`, so there is no savings. Nonetheless, the notion of a quasi-immutable value object based on `uvm_component` has merit. For instance, user code could build a hierarchical component structure of config objects that mirrors the hierarchical structure of configured components.

IMMUTABLE VS. RANDOM

Randomization is mutation, so immutable classes must not have random variables. However, most testbenches rely on randomization, so the question is how to create an immutable object with random values. One solution is to create another class: a randomizable `uvm_object`-based mutable counterpart to the immutable class.

Returning to our example, the mutable version of `box_config` is called `box_config_mutable`. This class implements `box_config_interface` to ensure polymorphic compatibility with `box_config_immutable`. Class `box_config_mutable` differs from `box_config` in the following ways:

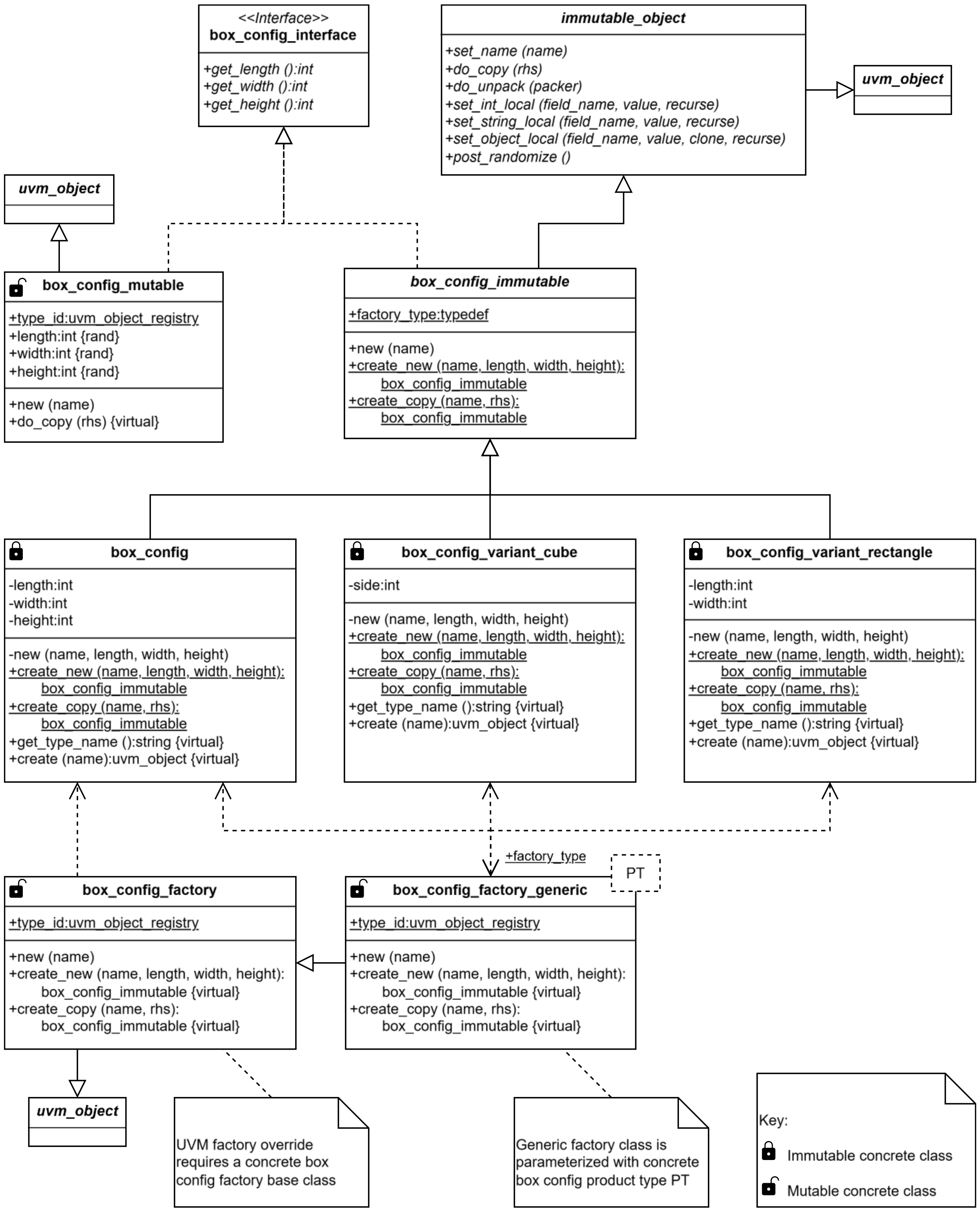
- It extends `uvm_object` instead of `immutable_object`.
- Its instance variables are public and random (keyword `rand`) instead of private (`local`) and fixed.
- Its constructor is public instead of `local`.
- Its setter functions are public instead of `local`.
- It is registered with the UVM factory with `` `uvm_object_utils_* `` macros.
- It does not require the immutable field flags.

User code creates a temporary mutable object, randomizes it, then uses the immutable class' `create_copy()` function or equivalent to create a new immutable object with the temporary object's values.

```
box_config_mutable temp = box_config_mutable::type_id::create("temp");
void'(temp.randomize());
box_cfg = box_config::create_copy("box_cfg", temp);
```

The shared `box_config_interface` implementation makes it simpler to copy values back and forth between `box_config` and `box_config_mutable` objects. The two classes have much behavior in common, so the developer can use a variety of OOP strategies, such as inheritance and composition, to reuse code between them.

Fig. 3 depicts a class diagram of the complete `box_config` ecosystem, including `box_config_mutable`.



<<Interface>>
box_config_interface
+get_length ():int
+get_width ():int
+get_height ():int
immutable_object
+set_name (name)
+do_copy (rhs)
+do_unpack (packer)
+set_int_local (field_name, value, recurse)
+set_string_local (field_name, value, recurse)
+set_object_local (field_name, value, clone, recurse)
+post_randomize ()
uvm_object
uvm_object
box_config_mutable
+type_id:uvm_object_registry
+length:int {rand}
+width:int {rand}
+height:int {rand}
+new (name)
+do_copy (rhs) {virtual}
box_config_immutable
+factory_type:typedef
+new (name)
+create_new (name, length, width, height): box_config_immutable
+create_copy (name, rhs): box_config_immutable
box_config
-length:int
-width:int
-height:int
-new (name, length, width, height)
+create_new (name, length, width, height): box_config_immutable
+create_copy (name, rhs): box_config_immutable
+get_type_name ():string {virtual}
+create (name):uvm_object {virtual}
box_config_variant_cube
-side:int
-new (name, length, width, height)
+create_new (name, length, width, height): box_config_immutable
+create_copy (name, rhs): box_config_immutable
+get_type_name ():string {virtual}
+create (name):uvm_object {virtual}
box_config_variant_rectangle
-length:int
-width:int
-new (name, length, width, height)
+create_new (name, length, width, height): box_config_immutable
+create_copy (name, rhs): box_config_immutable
+get_type_name ():string {virtual}
+create (name):uvm_object {virtual}
+factory_type
PT
box_config_factory
+type_id:uvm_object_registry
+new (name)
+create_new (name, length, width, height): box_config_immutable {virtual}
+create_copy (name, rhs): box_config_immutable {virtual}
box_config_factory_generic
+type_id:uvm_object_registry
+new (name)
+create_new (name, length, width, height): box_config_immutable {virtual}
+create_copy (name, rhs): box_config_immutable {virtual}
uvm_object
UVM factory override requires a concrete box config factory base class
Generic factory class is parameterized with concrete box config product type PT
Key:
Immutable concrete class
Mutable concrete class

Figure 3. UML class diagram of the `box_config` ecosystem.

Composing a Sequence Item

UVM's uvm_sequence_item base class is for modeling transactions that pass through the DUT. A sequence item should not be immutable. Rather it is more akin to an entity in that it has a singular identity and naturally mutates over its lifespan. However, a sequence item can and should be composed partially of immutable value objects. For example, a bus transaction's sequence item might contain:

- An immutable bus request object (read/write, size, address, write data, etc.)
- An immutable bus response object (read data, error response, etc.)

Fig. 4 illustrates a proposed sequence item structure in the context of its interactions with a generic sequence, sequencer, and driver. The figure shows a third member in the sequence item: a mutable, randomizable random_bus_req value object whose sole purpose is to support randomization of the sequence item with traditional `` `uvm_do_* `` and `` `uvm_rand_send_* `` macros. The random_bus_req object has random fields for bus protocol values, and within the sequence item, random_bus_req is itself declared as rand. As a result, the user's sequence can create and randomize the sequence item with a standard macro, which in turn recursively randomizes the mutable bus request object with optional in-line constraints. The developer can use post_randomize() to copy the newly randomized request into the sequence item's immutable request slot automatically. The mutable request has fulfilled its purpose and should not be used anymore for the lifetime of the sequence item. The sequence sends the sequence item with a populated bus request and empty bus response to the driver via the sequencer. The driver gets the item and drives the bus request on the bus. Then the driver samples the bus response, and stores it as an immutable response object in the appropriate slot in the sequence item, where the initiating sequence can retrieve it when the transfer completes, and react accordingly.

Fig. 5 illustrates similar connections between input and output monitors and a scoreboard. No mutable bus objects are required since monitors don't typically generate random sequence item contents; they populate them based on observed bus values. The monitors collect transactions from the DUT's interfaces, create sequence items with immutable request and response objects, and publish them through analysis ports to the scoreboard for checking.

This modular sequence item design introduces new classes, but it offers advantages. Encapsulating bus request and response objects in separate classes accurately models the directionality of the bus protocol; the transfer initiator owns the request, and the transfer target owns the response. The two classes enjoy tight cohesion and separation of concerns. Breaking up a monolithic sequence item into smaller, simpler classes with cleaner interfaces encourages reuse and unit-testing, leading to fewer defects. Object immutability underscores the creator/consumer relationship between components, and adds to model fidelity because the requests and responses represent snapshots of historical state. Subsequent events cannot change history. Testbench components like sequencers, drivers, monitors, and scoreboards run as concurrent threads, so any mutable object references that they share are susceptible to hazards like corruption and race conditions that typically require synchronization solutions. Immutable objects do not carry those same risks.

Conclusion

There is a widely accepted computing maxim, supported empirically by [10], that value objects should be immutable, primarily to avoid aliasing bugs. Certain IC verification concepts such as stable component configurations and data transfer snapshots benefit from being designed as value objects. Therefore, in a testbench, such items should be modeled as immutable objects. Effective DDD encompasses both design and process. Once the design decision has been made to incorporate immutable objects, this paper tackles the process question of how to implement them.

The SystemVerilog language, UVM idioms, and established verification techniques present obstacles to immutability. This paper identifies and explains the obstacles as they are encountered, and offers solutions that aspire to be broadly enabling and rigorously functionally correct, with minimal concern for cost, however that might be measured. As a result, the solutions are somewhat overengineered, but they reveal the contours of the problem space.

Armed with an understanding of the ramifications, the testbench designer can evaluate tradeoffs when considering different approaches for implementing immutable objects. If the designer doesn't need UVM factory overrides, then secondary factories aren't required. If the designer can live with the risks posed by extending immutable classes, then public constructors can be used and static factory methods omitted. If the immutable class doesn't need to descend from uvm_object, the designer can create a simpler SystemVerilog class instead. Even the H.A.C.K.S. may be viable. Design decisions will be driven by the requirements of each application.

Reference [15] is a public GitHub repository that contains complete functioning source code for the box_config and H.A.C.K.S. examples in this paper. In addition, it contains files from the UVM 1.2 reference implementation UBUS example with sequence items reworked with immutable bus transaction objects. Lastly, it contains a free open-source SystemVerilog package with the immutable_object base class and other utilities, which can be used as-is in any UVM project.

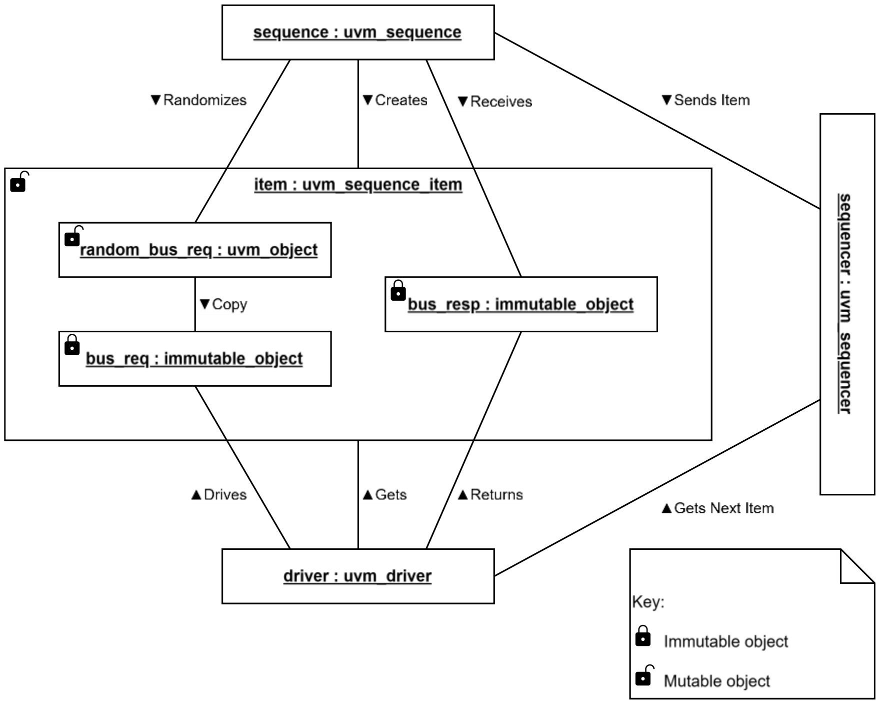


Figure 4. UML object diagram of a sequence and driver sharing a sequence item composed of mutable and immutable value objects.

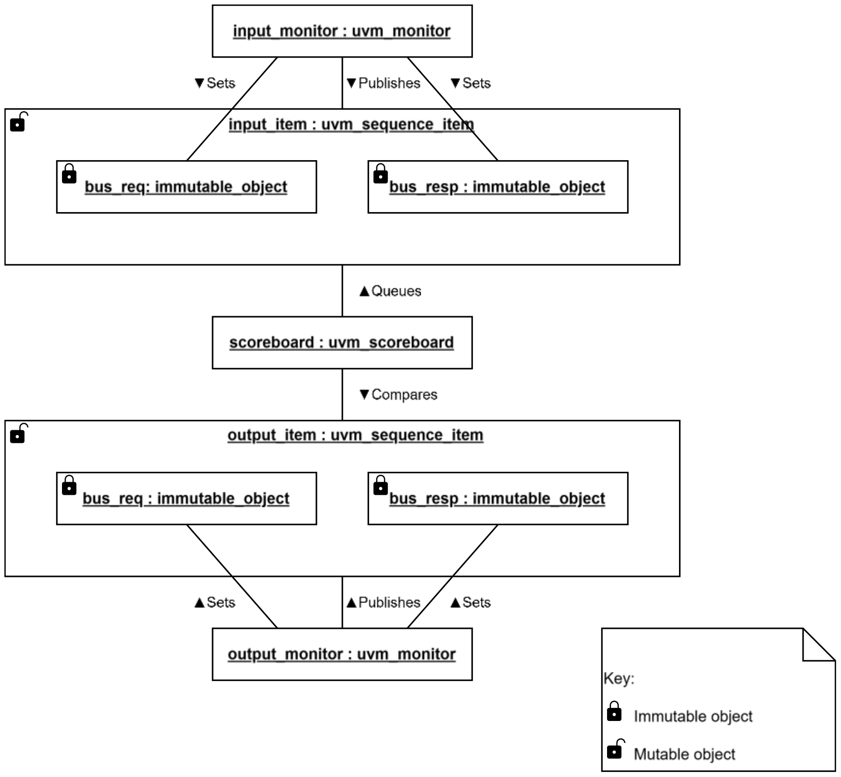

input_monitor : uvm_monitor
▼Sets
▼Publishes
▼Sets
input_item : uvm_sequence_item
bus_req: immutable_object
bus_resp : immutable_object
▲Queues
scoreboard : uvm_scoreboard
▼Compares
output_item : uvm_sequence_item
bus_req : immutable_object
bus_resp : immutable_object
▲Sets
▲Publishes
▲Sets
output_monitor : uvm_monitor
Key:
Immutable object
Mutable object

Figure 5. UML object diagram of monitors and a scoreboard sharing a sequence item composed of mutable and immutable value objects.

References

[1] Joel, B., "Just the Way You Are," Columbia Records, 1977.

[2] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800™-2012, 2013, p.190.

[3] *Universal Verification Methodology (UVM) 1.2 Class Reference*, UVM 1.2, Accellera, Napa, CA, USA, Jun. 2014.

[4] *Universal Verification Methodology (UVM) 1.2 User's Guide*, Accellera, Elk Grove, CA, USA, Oct. 2015.

[5] Gamma, E., Helm, R., Johnson, R., & Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[6] Glasser, M., "Next Level Testbenches: Design Patterns in SystemVerilog and UVM," Self-Published, 2024.

[7] Meade, K. and Rosenberg, S., "A Practical Guide to Adopting the Universal Verification Methodology," Cadence Design Systems, 2010.

[8] Evans, E., "Domain-Driven Design: Tackling Complexity in the Heart of Software," Addison-Wesley, 2003.

[9] MIT 6.005, "Immutability," MIT OpenCourseWare. [Online]. Available: https://web.mit.edu/6.005/www/fa15/classes/09-immutability/. [Accessed: 3-Nov-2024].

[10] C. Marsavina, "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems," 2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI), Timisoara, Romania, 2020, pp. 000105-000110, doi: 10.1109/SACI49304.2020.9118816.

[11] Oracle, "Immutable Objects," The Java™ Tutorials. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html. [Accessed: 3-Nov-2024].

[12] Fowler, M., "Patterns of Enterprise Application Architecture," Addison-Wesley Professional, 2003, Chapter 18.

[13] "Value Objects Should Be Immutable," C2 Wiki. [Online]. Available: https://wiki.c2.com/?ValueObjectsShouldBeImmutable. [Accessed: 7-Apr-2023].

[14] "Immutables," GitHub Repository. [Online]. Available: https://immutables.github.io/. [Accessed: 3-Nov-2024].

[15] W. L. Moore, "Immutables," GitHub Repository. [Online]. Available: https://github.com/williaml33moore/immutables. [Accessed: 3-Nov-2024].