Applying Design Patterns to accelerate development of reusable, configurable and portable UVCs.

accellera SYSTEMS INITIATIVE

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

About the presenter…

Paul Kaunds

- Paul Kaunds is a Verification Consultant at EnSilica Ltd, with over 16 years experience in the industry.
- Extensive knowledge of advanced verification (including SystemVerilog, UVM, eRM) and the use of new methodologies, tools and flows to deliver the best possible verification solution.

EnSilica Ltd

- EnSilica was founded in 2001 and has a strong track record of success in delivering semiconductor IP and providing ASIC and FPGA design services to semiconductor companies and OEMs worldwide. The company is a specialist in low-power ASIC design and complex FPGA-based embedded systems, including hardware and embedded software development.
- Our portfolio of IP includes eSi-RISC, a highly configurable 16/32 bit embedded processor and families of IP covering communications, processor peripherals and encryption
- For further information about EnSilica, visit http://www.ensilica.com.

Design Patterns

"Each pattern describes a problems which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution again a million times over, without ever doing it the same twice"

Christopher Alexander (Architect),
A Pattern Language: Towns, Buildings, Construction, 1977

Design Patterns

"The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing."

Christopher Alexander (Architect),
A Pattern Language: Towns, Buildings, Construction, 1977

What is Design Patterns

- A Design Pattern systematically names, explains, and implements an important recurring design.
- These are define well-engineered design solutions that practitioners can apply when crafting their applications.

Design Patterns

- A pattern is a proven solution to a problem in a context.
- Christopher Alexander says each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution.
- Design patterns represent a solutions to problems that arise when developing UVC within a particular context.

- **Patterns = problems.solution** pairs in a context

Design Patterns

- A Design pattern is a recurring solution to a standard problem, in a context.

- **Design patterns are Thought Processes.**

- **Design Pattern is like Dress Patterns**
- Jim Coplein, a software engineer:
  "I like to relate this definition to dress patterns…"

- ".. I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself."

Design Patterns History

- Started in 1987 by Ward Cunningham and Ken Beck who were working with Smalltalk and designing GUIs.
- Popularized by Gamma, Helm, Johnson and Vlissides (The gang of four, Go4)
- Design pattern use a consistent documentation approach
- Design pattern are granular and applied at different levels such as frameworks, subsystems and systems
- Design patterns are often organized as creational, structural or behavioral

Design Patterns elements

Design patterns have 4 essential elements

- Pattern name: increases vocabulary of designers
- Problem: intent, context, when to apply
- Solution: UML-like structure, abstract code
- Consequences: results and tradeoffs

Why Design Patterns

- Good designers do not solve every problem from first principles. They reuse solutions.

- Practitioners do not do a good job of recording experience in Verification Environment design for others to use. Patterns help solve this problem

Design Patterns are NOT

They are NOT:

- Data structures that can be encoded in classes and reused as is (i.e., linked lists, hash tables)

- Complex domain-specific designs
(for an entire application or subsystem)

3 Type of Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their **Design Patterns** book define 23 design patterns divided into three types:

- **Creational patterns** are ones that create objects for you, rather than having you instantiate objects directly. This gives your Verification Environment more flexibility in deciding which objects need to be created for a given case.

- **Structural patterns** help you compose groups of objects into larger structures, such as complex UVM SOC Verification Environment

- **Behavioral patterns** help you define the communication between objects in your system and how the flow is controlled in a complex UVM SOC Verification Environment

Design Patterns Catalogue

CREATIONAL PATTERNS

1. **Factory Method**
2. **Abstract Factory**
3. **Builder**
4. **Prototype**
5. **Singleton**

STRUCTURAL PATTERNS

1. **Adapter**
2. **Bridge**
3. **Composite**
4. **Decorator**
5. **Façade**
6. **Flyweight**
7. **Proxy**

BEHAVIORAL PATTERNS

1. **Chain of Responsibility**
2. **Command**
3. **Interpreter**
4. **Iterator**
5. **Mediator**
6. **Memento**
7. **Observer**
8. **State**
9. **Strategy**
10. **Template Method**
11. **Visitor**

Benefits of Design Patterns

- Design patterns enable large-scale reuse of UVC architectures and also help document.
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Drawbacks of Design Patterns

- Patterns are validated by experience and discussion rather than by automated testing

Anti-Patterns

- Anti-patterns are certain patterns in Environment development that is considered as bad programming practice.

- As opposed to design patterns which are common approaches to common problems which have been formalized, and are generally considered a good development practice, anti-patterns are the opposite and are undesirable.

Anti-Patterns

- For example, in object-oriented programming, the idea is to separate the software into small pieces called objects. An anti-pattern in object-oriented programming is One Single huge object which performs a lot of functions which would be better separated into different objects.

Architecture Patterns Vs Design Patterns

- An architectural pattern is a general, reusable solution to a commonly occurring problem in architecture within a given context.
- Architectural patterns are similar to design pattern but have a broader scope
- An architectural pattern is a concept that solves and delineates some essential cohesive elements of a architecture.

Architecture Patterns Vs Design Patterns

- Design patterns: Solves reoccurring problems in software construction
- UVM Usecase: Factory Method, UVM Configuration Object.

- Architectural patterns: Fundamental structural organization for software systems
- UVM Usecase: UVM Agent, UVM Scoreboard

Object Oriented Programming Basics

- Abstraction

It is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics

- Encapsulation

It is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data.

- Polymorphism

It is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

- Inheritance

when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes.

Guiding Principles Of OOP

- Classes should be open for extension but closed for modification (Open Close Principle)
- Subclasses should be substitutable for their base classes

  (Liskov Substitution Principle)
- Depend on abstractions. Do not depend on concrete classes

  (Dependency Inversion Principle)
- Encapsulate what varies
- Favor composition over inheritance
- Loosely coupled designs between interacting objects

Open-closed Principle (OCP)

"*software entities like Classes should be open for extension, but closed for modification*"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

Meyer's open/closed principle

- Implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through inheritance. The derived subclass might or might not have the same interface as the original class.

Polymorphic open/closed principle

- Implementations can be changed and multiple implementations could be created and polymorphic ally substituted for each other.
- In Inheritance from abstract base classes, Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

Liskov Substitution Principle

- It states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may *substitute* objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)
- It defines a notion of substitutability for mutable objects;

Dependency Inversion Principle (DIP)

- It refers to a specific form of decoupling software modules.
- When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed), thus rendering high-level modules independent of the low-level module.
- *High-level modules should not depend on low-level modules. Both should depend on abstractions*
- *Abstractions should not depend on details. Details should depend on abstractions*

Encapsulate what varies

- Looks for behaviours that (may) change, e.g. different algorithms
- Encapsulate what changes in a class
- Aims to allow changes to be made without affecting dependent code

Design Patterns

"Each pattern describes a problems which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution again a million times over, without ever doing it the same twice"

Christopher Alexander (Architect),
A Pattern Language: Towns, Buildings, Construction, 1977

Design Patterns

- Patterns tell us how to structure classes and objects to solve certain problems.
- We need to fit that to our application and programming language
- Embody Object Oriented Principles
- Show you how to write code with good Object oriented design qualities
- Most patterns address change

Design Patterns

Creational

- **Factory Method**

It Defines an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

Using this method, objects are constructed dynamically based on the specification type of the object. User can alter the behavior of the pre-build code without modifying the code. From the testcase or UVM Commandline, user can replace any object which is at any hierarchy level with the user defined object.

This is mainly for UNPLANNED Changes in UVM. Configuration is used for planned changes.

How Architecture reacts to Changes is very important from reusability perspective.

Factory + Configurations = Flexibility.

Design Patterns

Creational

- **Singleton**

Ensure a class has only one instance, and provide a global point of access to it.

Benefits:

- Controlled access to sole instance
- Reduced name Space
- Permits refinement of operations and representation

UVM Usecases: UVM Pool, UVM Resource Pool, UVM Global report server, UVM command line processor

Design Patterns

Creational

- **Abstract Factory**

Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.

UVM Usecases: Polymorphic Interfaces, Abstract Checker Environment

Design Patterns

Structural

- **Composite**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

UVM Usecases: UVM Component Hierarchy- UVM Env, UVM Sequence Library-UVM Sequence , UVM Configuration

Factory + Configuration = Flexibility

Design Patterns

Structural

- **Facade**

The name is by analogy to an architectural facade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

A facade can

- make a software library easier to use, understand and test, since the facade has convenient methods for common tasks;
- make the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;

UVM Usecases: TLM Ports. TLM Ports used in UVM Agent, UVM Scoreboards

Design Patterns

Structural

- **Adapter**

Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.

UVM Usecases: UVM Reg Adapter

Design Patterns

Structural

- **Bridge**

Decouple an abstraction from its implementation so that the two can vary independently.

UVM Usecases: UVM Sequencer and UVM Driver

Design Patterns

Behavioral

- **Observer**

Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

UVM Usecases: UVM Subscriber, UVM Monitor, UVM Coverage, UVM Scoreboards.

Design Patterns

Behavioral

- **Template Method**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

UVM Usecases: UVM Transaction (do_copy, do_compare), UVM Phase

Design Patterns

Behavioral

- **Command**

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

UVM Usecases: UVM Sequence Item

Design Patterns

Behavioral

- **Strategy**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

UVM Usecases: UVM Sequence, UVM Driver

Design Patterns

Behavioral

- **Mediator**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other and it lets you vary their interaction independently.

UVM Usecases: UVM Virtual Sequencer

Abstract Factory Pattern (Creational)

Problem:

I want to instantiate different versions of Interface protocol like USB/PCIe, depending on the Product being used. I don't want my environment change when new version of Interface protocol are added.

Solution: Abstract Factory

Provides and interface for creating families of related or dependent objects without specifying their concrete classes

Example: 1. Configurable Bus Functional Models-Polymorphic Interfaces. 2. Polymorphic Interface binding used in UVM Test harness makes block level environment reusable at multi-block or system level verification Environments.

Composite Pattern (Structural)

Problem:

I want to build a tree structure, where objects can be leafs or other nodes to build flexible configurable systems.

Solution: Composite Pattern

Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects or compositions of objects uniformly

Example: UVM Env Hierarchy, UVM Configurations.

Strategy Pattern (Behavioural)

Problem:

I have code that uses an algorithm that can change or new ones can be added. I want to allow the algorithm to change without breaking my code.

Example: Moving from USB 2.1 to USB 3.0 or PCIe 3.0 to PCIe 4.0

Solution: Strategy Pattern

Defines a family of algorithms, encapsulates each one and makes them interchangeable.

Strategy lets the algorithm vary independently from the system using it.

Example: 1. UVM Sequence - Protocol specific stress/congestion scenarios.

2. UVM Driver – Conversion of Transactions into Protocol specific frame formats at bit level.

References

1. Design patterns : elements of reusable object-oriented software by ErichGamma, RichardHelm, RalphJohnson, JohnVlissides
2. Flexible UVM Components: Configuring Bus Functional Models by Gunther Clasen, Ensilica

Questions