Object Oriented Programming

Puneet Goel

Consultant
Coverify Systems Technology

April 2017

In this section ...

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to the human beings what we want a computer to do.

Knuth, 1992

In this section ...

Rigidity

- Difficult to change design – even in simple ways
- Every small change cascades into subsequent changes in dependent modules
- When managers let loose their team on a required change in design, the engineers disappear for a long time
  - As a result, the managers try to block any request for a change in requirement or architecture

Fragility

- Every change, however small, breaks the design in unexpected ways
- A small bug-fix may result in a barrage of unrelated bugs, many of them getting resulting from the fix, but getting caught much later
- The management and the customers loose trust on the development team

Immobility

- Inability to reuse the design in other projects or parts of the design at other places in the same project
- The design may have been too tightly integrated with other modules, thus forming a baggage that is too difficult to remove
- May result from:
  - Poor encapsulation
  - Not following standard coding practices – even rules tha might sound trivial, such as naming conventions

Viscosity

- A design/development environment is termed viscous when it is easier to pull in changes via hacks compared to the methodology
- Viscosity can be in the design or in the development environment
- Viscosity may result in employment of non-standard ways, often resulting in more viscous systems at a later stage
- A major source of viscosity in chip industry stems from lack of interest in software tools (like the version control)

Changes in Requirements

- Many of the Software Perils that we discussed, result from changing requirements
  - Requirements often change in a way not anticipated by the designers
  - Requirement document is the most volatile document and we must realize this and plan ahead for possible changes in requirements

In this section . . .

Agile Programming

One Solution to all Our Problems

- ~~Rigidity~~
- ~~Viscosity~~
- ~~Immobility~~
- ~~Fragility~~

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over Processes and tools
- **Working software** over Comprehensive documentation
- **Customer collaboration** over Contract negotiation
- **Responding to change** over Following a plan

That is, while there is value in the items on the right, we value the items on the left more.

In this section . . .

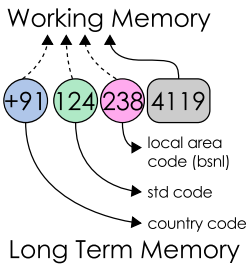
The Magical 7 +/- 2

Have you heard about the *Magical Number Seven Plus or Minus Two*?

- Numerous experiments have shown that a human mind can *actively manage* about seven pieces of information. This capacity of human mind is known as *working memory*.
- Given the limited capacity of the human mind, how can we go about making complex systems?
  - Create complex, incomprehensible systems that you will not understand yourself when you revisit it after 6 months.
  - Use Chunking.

The Magical 7 +/- 2 (2)

chunking makes use of long term memory



- *Chunking* refers to a strategy for making more efficient use of short-term memory by recoding information.
- A *Chunk* indicates a long-term memory structure that can be used as units of perception and meaning.

The Magical 7 +/- 2 (3)

Chunking to chess grand-masters rescue

Have you ever heard the story of the king offering to fulfill a wish of the discoverer of the game of chess? The number of various possible sequence of chess moves is too huge for a human minds capacity. How do the chess grand-masters manage?

- In the Figure, various positions of chess pieces have been *chunked* together to make groupings (including castling).
- This make it easier for Chess grand-masters to have an understanding of the game.

Chunking in Programming Languages

- Procedural programming languages such as ANSI C provide data grouping constructs that enable the user to manage complex data structures in a hierarchical manner.
- A programmer coding the higher protocols need not be aware of the structural details of the ethernet frame. He may just use a set of library routines to process the frames as per the lower layer protocols.

Ethernet Frame in C

```
struct Frame {
  unsigned char preamble[8];
  unsigned char dest_addr[6];
  unsigned char src_addr[6];
  unsigned char type_field[2];
  unsigned char* data;
  unsigned int data_size;
  unsigned char crc[4];
};

void gen_preamble
(struct Frame* fr);
void compute_crc
(struct Frame* fr);
void print_frame
(struct Frame fr);
```


Object-based programming

Object-based languages allow grouping of functions pertaining to a given `struct` along with the other *attributes* of the `struct`. The *attributes* may actually be hidden from the user, allowing access to them only through the qualified *methods*. These methods are often termed the *user interface* to the class.

Ethernet Frame in C++

```
class Frame {
  public: // user interface
  Frame(unsigned int size=128);
  void compute_crc();
  void print_frame();
  virtual unsigned int byte_pack
  (std::vector<unsigned char>& bytes);
  private: // hidden
  unsigned char preamble[8];
  unsigned char dest_addr[6];
  unsigned char src_addr[6];
  unsigned char type_field[2];
  std::vector<unsigned char> data;
  unsigned char crc[4];
};
```


A Class is a Contract

- A class is a *contract*. The class interface (public methods) helps fulfill the contract.
- The private/local data members are implementation details. These could be used in an unauthorized manner if not declared as private.

The Single Responsibility Principle

Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

- A responsibility is a reason for *change*. A class or module should have one, and only one, reason to change.

The Open-Closed Principle

Software entities (classes, modules, functions, etc.) should be *open for extension*, but *closed for modification*; that is, such an entity can allow its behaviour to be modified without altering its *source code*.

Class Inheritance

Inheritance in C++

```
class CryptedFrame: public Frame {
  public:
  virtual unsigned int byte_pack
  (std::vector<unsigned char>& bytes) {
    unsigned int size =
    Frame::byte_pack(bytes);
    // Encrypt the packed data
    encrypt(bytes);
    return size;
  }
  private:
  // Auxiliary functions
  void encrypt
  (std::vector<unsigned char>& bytes);
  // Auxiliary data
  unsigned long encrypt_key;
};
```


Class Inheritance (2)

- Inheritance is used to express 'is a' relationships.
- Unlike other class methods, a virtual function is bound at run time instead of at compile time.
- We cannot instantiate objects from virtual classes – (in C++ classes with pure virtual functions).
- In C++, class objects have to be passed by reference (or as pointers), if we do not wish to change the virtual functions that get invoked.
- Virtual functions are implemented using a table of functions pointers.

Non-OCP code leads to Rigidity

Example code without OCP

```
struct Modem {
  enum Type {hayes, courrier, ernie} type;
};

struct Hayes {
  Modem::Type type;
  // Hayes related stuff
};

struct Courrier {
  Modem::Type type;
  // Courrier related stuff
};

void LogOn(Modem& m,
           string& pno, string& user, string& pw)
{
  if (m.type == Modem::hayes)
    DialHayes((Hayes&)m, pno);
  else if (m.type == Modem::courrier)
    DialCourrier((Courrier&)m, pno);
  // ...you get the idea
}
```

- Whenever a new modem requires to be added, the code for LogOn will need updation

How OCP helps tackle Rigidity

Example code with OCP

```
class Modem
{
public:
  // LogOn has been closed for modification
  virtual void Dial(const string& pno) = 0;
  virtual void Send(char) = 0;
  virtual char Recv() = 0;
  virtual void Hangup() = 0;
};
void LogOn(Modem& m,
           string& pno,
           string& user,
           string& pw)
{
  m.Dial(pno);
  // you get the idea.
}
```

- The function LogOn has been closed for modification – Other functions like the Dial function have been abstracted out

OCP using static Polymorphism

```
template <typename MODEM>
void LogOn(MODEM& m,
           string& pno,
           string& user,
           string& pw) {
  m.Dial(pno);
  // you get the idea.
}
```


The Liskov Substitution Principle

Or how to find out if two classes should have "is a" relation

Liskov's notion of a behavioral subtype defines a notion of substitutability for mutable objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program

Question

From LSP perspective, *is a* square a rectangle? Is it a good idea to derive a Square Class from a Rectangle Class?

The Liskov Substitution Principle (2)

- A typical example that violates LSP is a Square class that derives from a Rectangle class, assuming getter and setter methods exist for both width and height.
  - The Square class always assumes that the width is equal with the height. If a Square object is used in a context where a Rectangle is expected, unexpected behavior may occur because the dimensions of a Square cannot (or rather should not) be modified independently.
- This problem cannot be easily fixed:
  - if we can modify the setter methods in the Square class so that they preserve the Square invariant (i.e. keep the dimensions equal), then these methods will weaken (violate) the postconditions for the Rectangle setters, which state that dimensions can be modified independently.

The Liskov Substitution Principle (2)

- Clients ruin everything that we built!
- When we think about the LSP, also think about the client interfaces:
  - Does the derived type (Square) have the same interface as the base type (Rectangle)?

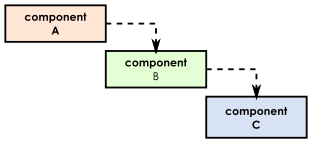
IS A Vs HAS A

Two objects are said to be related to each other by a *has a* relation, if one object is *aggregated* by the other object through composition.

- More often what you need is a *has a* relation than *is a*. Inheritance is often misused.
- Both the relation types are similar in one aspect – both provide *containment*.
- In some ways, a *has a* relation can provide you all the functionality of *is a* relation.

- Since SystemVerilog does not enable multiple inheritance yet, consider *has a* type containment when you need multiple inheritance.

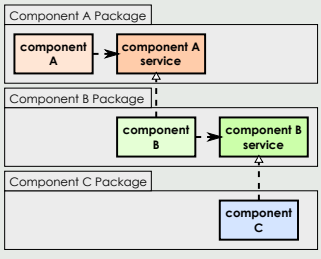
Fragility: Tight Binding



- As the software is developed using a layered approach, the upper layers often need to call the functions and methods made available by the lower layers.
  - For example a *utility* layer/library for a modem related application could provide parity and CRC functions. A higher level protocol layer would need to call these methods to check the parity/CRC of the data received on an ingress channel.
- This results in tight binding – with the upper level layers becoming dependent on the lower level layers.

Fragility: Tight Binding (2)

The Solution – Dependency Inversion Principle



Dependency Inversion Principle

The principle suggests that . . .

- **High-level modules should not depend on low-level modules. Both should depend on abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions.**

The Interface Segregation Principle

Many client specific interfaces are better than one general purpose interface
If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multi and multiply inherit them into the class.

- If clients share interfaces, a change in interface for one client would spoil everything Do you see Rigidity here?

What is Generic Programming about?

Question What are the three basic tenets of (Object Oriented) Programming?

1. Abstraction
2. *Abstraction*
3. **Abstraction**

- Actually programming is all about **reuse**. Let is see how!

Abstracting out Magic Numbers

Magic numbers in C++

```
// ....
area = 3.141592458 * radius * radius;
// ....
circumfrence = 3.141592458 * radius * 2;
// ....
```

Using constants in C++

```
// ....
const double PI = 3.141592458;
area = PI * radius * radius;
// ....
circumfrence = PI * radius * 2;
// ....
```

- Which code is preferred and why?
- Do you see how constants are abstracted out in the second example?

Abstracting out Algorithms

Water Fall coding in C++

```
// ....
area1 = 3.141592458 * radius1 * radius1;
// ....
area2 = 3.141592458 * radius2 * radius2;
// ....
area3 = 3.141592458 * radius3 * radius3;
// ....
```

Using functions in C++

```
// ....
double get_area(double radius)
{
  const double PI = 3.141592458;
  return PI * radius * radius;
}

area1 = get_area(radius1);
// ....
area2 = get_area(radius2);
// ....
area3 = get_area(radius3);
// ....
```

- Notice how functions help abstracting out algorithms
- Abstraction helps in reuse

Abstracting out Algorithms (2)

Water Fall coding in C++

```
// sort in increasing order
void sort(// ....)
{
  // ....
  if (x < y) // ....
  else // ....
  // ....
}
```

Using functions in C++

```
// generic sort
void sort(// ....
bool (* isOrdered)(double, double))
{
  // ....
  if(isOrdered (x, y)) // ....
  else // ....
  // ....
}
```

- Pointers are complicated! Any volunteers who can help us understand the declarations here?
- Here we are passing algorithm as an argument to another!
- Functors are also useful in such scenarios

The third level of abstraction

Typed and Untyped Languages

How can we achieve still more abstraction?

- Both C++ and SystemVerilog are *statically typed* languages. This basically means that the parameter values passed to a function are checked against the argument types at the time of linking.
- Static type checking leads to . . .
- Less run-time crashes – type errors are caught at the time of compilation itself
- Faster code – no need to check the type at run-time
- Less generic code – functions need to be coded again for each type the functionality is required.

Static Polymorphism

- Both C++ and SystemVerilog are statically type-checked – meaning thereby, the algorithms we code as functions can not be applied in a generic fashion.
- Let us take a look at a trivial example that returns maximum of two arguments.

Max for integers in C++

```
// Returns Maximum of two arguments
int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}
```


Static Polymorphism (2)

- Because the arguments are checked for types, the code that we have written can not be applied to anything but `int`s
- If we want to write a `max` function for `double`s, we shall have to write another function . . .

Max for doubles in C++

```
// Returns Maximum of two arguments
double max(double a, double b)
{
  if(a > b)
    return a;
  else
    return b;
}
```

- Hmmm . . . Do you see an opportunity for *code reuse*.

Static Polymorphism (3)

- We have seen how static polymorphism enables algorithm reuse by making it possible to write generic functions. What about data structures?
- Both SystemVerilog and C++ make it possible to code generic classes. Let us illustrate by way of an example . . .

Generic Complex Numbers

```
1  template <class T>
2  class Complex {
3    public:
4    Complex(T realp); // Constructor
5    Complex(T realp, T imagp);
6    // other functions ...
7    private:
8    T realp_;
9    T imagp_;
10 };
```


Dynamic Polymorphism

- Till now we have seen how templates help us write generic (type independent) functions and data-structures.
- There are situations where we might need to work with a collection of objects of varied types – such as a collection of shapes.
  - In the chip verification domain we would like to have a stream of packets of different types of ethernet frames.
  - Or ATM frame packets – with some of these packets having CRC error purposefully introduced.
- It is easy to implement such collections in untyped languages. But it is also easy to create run-time errors in such languages.
- Typed OOP languages such C++, enable such collections, if the objects are related by way of inheritance – actually if the objects have a common parent.

Dynamic Polymorphism (2)

Let us see some example code to understand how such collections are made possible in C++:

C++ Collection of Shapes

```
1  class Shape {
2    public:
3    // ....
4    private:
5    Color color_;
6  };
7  class Circle: public Shape {
8    public:
9    // ....
10   private:
11   Point center_;
12   double radius_;
13 }
14 class Square: public Shape {
15   public:
16   // ....
17   private:
18   Point sw_corner_;
19   double side_;
20 }
21 // ....
22 std::vector<Shape*> shapes;
```


In this section . . .

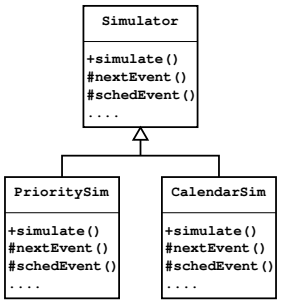
What Patterns shall We Learn

- Template Method Pattern
- Strategy Pattern
- Facade Pattern
- Singleton Pattern
- Factory Pattern
- Observer Pattern

Template Method Pattern

- Remember the Dependency Inversion Principle?
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend upon details. Details should depend upon abstractions.*
- In order to confirm to DIP, we want to make sure that generic algorithms do not depend on detailed implementation
  - Rather generic algorithms and detailed implementations should both depend on abstractions

Template Method Pattern



- The `simulate` method is fully defined in the base class `Simulator`
- Depending on the chosen scheduler algorithm, implementation of `nextEvent` and `schedEvent` methods are picked

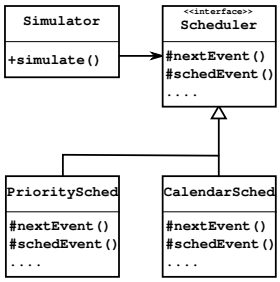
Strategy Pattern

Strategy Pattern is quite similar Template Method Pattern:

- Template Method Pattern uses inheritance
- Strategy Pattern uses delegation for solving same problem

Favor object composition over class inheritance – GOF95

Strategy Pattern



- Strategy Pattern uses delegation and hence allows more flexibility with the scheduler algorithm
  - It is now possible to choose the scheduler algorithm even after a `Simulator` instance has been created

The Facade Pattern

The Facade Pattern is used when you want to provide a simple and specific interface onto a group of objects that has a complex and general interface

For example:

- The PLI interface to the verilog simulator
- VMM provides `byte_pack()` function as an interface function between the different layers of verification platform

Singleton Pattern

- Most objects are dynamic. Mulitple objects are created and destroyed in the life of an application.
- But sometimes a class object might be of static nature; it needs to be created right when the application is started and gets destroyed when the application is closed. And there is only a single instance of such cleass required.
- Examples include – factories – managers (like the window manager) etc

Singleton Pattern

singleton.d

```
1  public class singleton
2  {
3    private static Singleton
4                   theInstance = null;
5
6    private this() {}
7
8    public static Singleton Instance()
9    {
10     if (theInstance == null)
11       {
12         theInstance = new Singleton();
13       }
14     return theInstance;
15   }
16 }
```

- Why do you need a pattern like the Singleton? If you need a single instance of a class, just make sure that you do that!
- Remember classes are contracts? Singleton forces a contract obligation on the class client.

The Factory Pattern

The Dependency Inversion Principal asks you to **not** depend on a class

- The principal asks you to depend on an interface instead

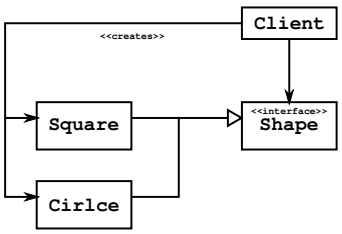
The problem is “How to create an object of the class?”

- Creating an object of the class requires dependency on the concrete class
- Interfaces are abstract – instances of interface can not be created

The Factory Pattern

factory1.d

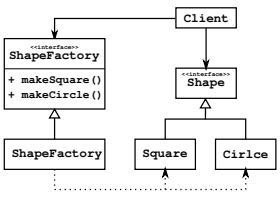
```
1  import std.stdio;
2  abstract class Shape {
3    void draw();
4    double area();
5  }
6  class Circle: Shape {
7    final override void draw() {
8      writeln("Drawing a Circle");
9    }
10   final override double area() {
11     return 3.14159;
12   }
13 }
14 class Square: Shape {
15   final override void draw() {
16     writeln("Drawing a Square");
17   }
18   final override double area() {
19     return 4.0;
20   }
21 }
22 void main() {
23   Shape foo = new Square();
24   foo.draw();
25 }
```



The Factory Pattern

factory2.d

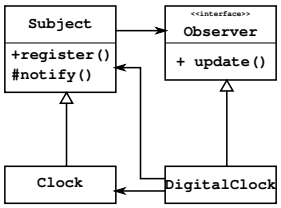
```
1  interface ShapeFactory {
2    public Shape makeCircle();
3    public Shape makeSquare();
4  }
5
6  class ShapeFactoryImpl: ShapeFactory {
7    public final Shape makeCircle() {
8      return new Circle();
9    }
10   public final Shape makeSquare() {
11     return new Square();
12   }
13 }
14
15 void main() {
16   ShapeFactory factory
17     = new ShapeFactoryImpl();
18   Shape foo = factory.makeSquare();
19   foo.draw();
20 }
```



The Factory Pattern

- The Factory pattern is often split into abstract factory and concrete factory patterns
- There can infact be multiple concrete factories corresponding to an abstract factory
- There are often other reasons for applying Factory Pattern
  - Creation of objects might be a quite complex and elaborate function for some classes
- Does Factory Pattern resolve rigidity?
  - Is that a complete solution?

The Observer Pattern



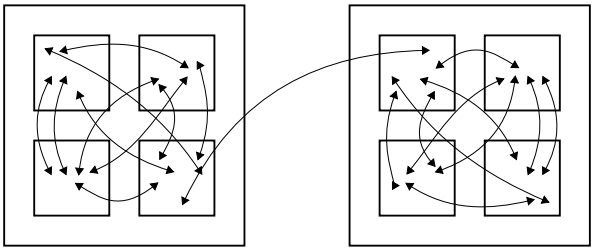
- DigitalClock observes Clock
- Clock registers DigitalClock instance as Observer
- When time changes a notification is sent to DigitalClock
- DigitalClock updates time

Observer Pattern and OCP

- The principal that drives Observer Pattern most is OCP
- An observer is supposed to observer its subject without any effort to change it
  - The observed object stays closed to modification
- In SystemVerilog, since the language does not enable `const` objects, a subject become prone to misuse

In this section . . .

Why Creating Packages is Important



While classes in the same package exhibit strong cohesion, there is hardly any dependece between classes belonging to two different packages.

Revisiting Encapsulation

- C++ does not have a package construct, but allows a user to declare classes as friends
- SystemVerilog programmers make all the data members of a transaction class public
- Generally as a rule, classes with strong cohesion between them should be included in one package
- And a package formed in this manner should be considered as a **reuse unit**

The Release Reuse Equivalency Principle

The granule of reuse is the granule of release

When you create a software component for reuse, consider making its release cycle independent. When a new release is made, a customer may not pick that release immediately. For a customer to reuse a package effectively, it is imperative that a reusable package is released as one unit and a release version tracking system be in place to maintain the package.

The Common Closure Principal

Classes that change together, should be grouped in the same package

When a class strongly relates to other classes, it will need changes every time any other class it relates to changes. In fact this ripple effect (changes in one class triggers changes in another) is often used as a metrics measure cohesiveness between two classes.
Since a package is used as a software release unit, it is imperative that classes that change together should be part of the same release package. Splitting such cluster of classes (that change together) into multiple packages, would necessitate that a new version of all these packages be released every-time there is a change in the cluster of classes, thus putting additional burden on the teams making the releases and on the integration teams.

The Common Reuse Principle

Classes that aren't reused together, should not be grouped in the same package

Suppose you get a package that contains many classes that you have no use of. Changes in any of these redundant classes would necessitate a new release of the package. If these redundant classes were not part of the package, you would not have to deal with these releases. More frequent package releases result in additional burden since the whole application goes through a cycle of tests every-time a new release of a package is integrated in the application.