International Journal of Computer Science and Informatics

Volume 2 | Issue 3 Article 9

January 2013

A Trenchant Analysis of Relationships in Object Oriented Domain Paradigms

Ajeet K. Jain
Computer Science & Engineering, Faculty of Science & Technology, jainajeet1@rediffmail.com

Seravana Kumar
ICFAI Foundation for Higher Education, Survey # 156 / 157, Dontanpalli(V), Shankarpalli Road, Hyderabad 501 203, seravanakumar@gmail.com

A Trenchant Analysis of Relationships in Object Oriented Domain Paradigms

Ajeet K. Jain & Seravana Kumar

Computer Science & Engineering, Faculty of Science & Technology
ICFAI Foundation for Higher Education, Survey # 156 / 157, Dontanpalli(V), Shankarpalli Road, Hyderabad 501 203
Email: jainajeet1@rediffmail.com & seravanakumar@gmail.com

Abstract **-** Object oriented paradigms provide a number of ways to permanently alter the software engineering field, catapulting it into the realm of true elegant design. Object oriented paradigms have taken the software evolution as a means of managing divergent complexities of development. The challenges through OO modeling / programming incorporating design paradigms are making head way for developing robust, reliable and maintainable software. The program code can be written, tested and modeled for reusability as a design process. The dynamic behavior modeling implementing the state models using OOAD and UML are most popular now with wider acceptance[1][2]. The importance of design paradigms and patterns are increasing ever fast in crafting complex systems. The software design patterns allow describing fragments and reuse of these design ideas in order to help the developers leverage. In this paper, the focus is aimed at addressing following issues:

- How OO relashionships can be used to form abstractions of higher derivatives adapting best practices
- Siutability of relationships among OO paradigms and their pertinent analysis
- How to map design artifacts incorporating languages like C++ and Java

Keywords **-** *Abstract classes, base classes, Is-A, Has-A, Part-of, multiple inheritance and interfaces, mixin classes, subclass, superclass*

I. INTRODUCTION

One of the most difficult aspect of OOP and OOD involves determining the organization of classes and objects. There are many different ways in which classes and objects can interact. Defining the relationships between the entities that may exist in a object oriented system makes it easy to understand and explore and thereby modify the system. There are various kinds of relationships depending upon the context and their interconnectness.

1.1 Is - A Relationship [3]

This is a specialized relationship which is used to indicate that one class is a variant of another class. Stated simply, " **Class B is – a Class A"** , indicates that the major characteristics of **class B** are inherited from **class A**.

Ex. **A** Banana **IS - A** **F**ruit or a Lunch Has- A Banana

// Class Two_Dim : public Shape { …. } ;

// Class Saving _Acct extends Account { }

1.2 Has – A Relationship

This is a containment relationship which is used to indicate that one class or object is a part of some other class or object. While **IS- A** relationship can only be used to define a relationship between classes, **HAS – A** relationship can be used to define the relationship between classes, between an objects and a class, or

between objects. Stating that "**Class B Has - A object A** " indicates that **object A** is a component of **class B**, i.e; **Class A** is used as a building block in construction of **class B**. The Is – A relationship can be defined with inheritance and Has – A relationship is well associated with composition.

Ex: **class Building { … }**

```
class House extends Building
 { …      }
class House
 {
   Dinninghall room = new Dinninghall();
    ……..
   public void getWindows()
     {
        room.getWindowSize();
       ……..          }
 }
```


1.3 Uses – A Relashionship

This is a **Using** relationship which indicates that a member function of one class accepts and therefore uses , an object of some other class as a parameter. For example, stating that " **class B USES – A class A** " indicates that **class B** objects uses the facilities offered by class A object , not that class A used as a building block in the construction of class B.

Ex.**: class Employee { }**

```
class HealthInsurance
  {
      Attributes:
       issuer
      Methods:
       //constructor
       HealthInsurance()
      {
          this.issuer=issuer;
       }
}
me = new employee();
myHealthCard= new HealthInsurance(me);
```

By the same token, a Computer can use a Laser printer, but it does not make sense to define a **Printer** class from a **Computer** class or v/v. We might however, devise a friend function or classes to handle the communication between Printer ojects and Computer objects. (public inheritance does not model this relashionship)


1.4 Is – Like – A Relashionship

This relationship does not do simile, (i.e.; comparing two unlike properties or things), which do not share any common properties, like:

Van is like a Car

Lawyers are like Sharks

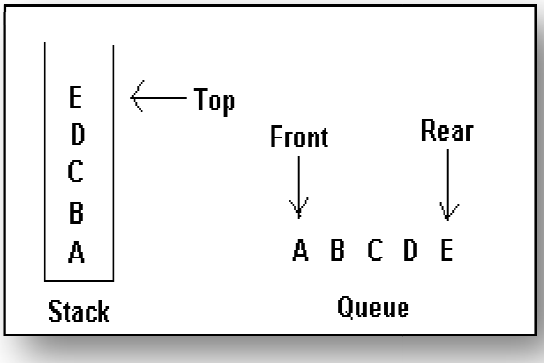
Human are Animal

But it is not true that a lawer is a shark or a lying person is a polytician. For insatance, sharks can live under water – what about lawer? Therefore, we should not derive aLawer class from a Shark class.

Inheritance can add properties to a base class; it does not remove properties from the base class. In some cases, shared characterstics canbe handled by designing a class encompassing those characteristics and then using that class, either in an **HAS-A** or **IS-A** relashionship, to define the related classes.

1.5 IS- IMPLEMENTED –AS-A Relashionship

This relashionship could be implementation of **Stack** using an array. However, it won't be proper to derive a **Stack** class from an **Array** class. A stack is not an array. For instance, array indexing is not a stack property. Also, a stack could be implemented in some other way, such as using a linked list or que. A proper approcah could be to hide the array implementation by providing the **Stack** a **private** object member (public inheritance does not model this relashionship).



1.6 A -Kind- Of Rlationship

This kind of relashioship is impelmented when we write a drawing program that would allow drawing of various ***objects*** such as **points**, **circles**, **rectangles**, **triangles** and many more. For each object we can provide a **class** definition; like, the **Point** class just defines a point by its coordinates:

```
class Point {
int  x, y ;
functions()  or methods()
  setX ( int newX )
  getX()
  setY(int newY)
  getY()
}
```

We can define classes of our drawing program with a class to describe circles, which defines a center point and a radius:

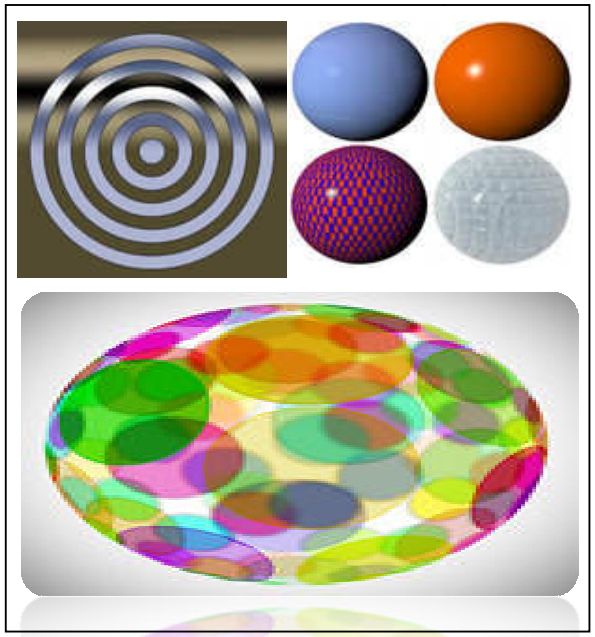
```
class Circle {
  int x, y,   radius ;
// methods
  setX(int newX)
  getX()
  setY(int newY)
  getY()
  setRadius(newRadius)
  getRadius()
}
```

Comparing the above two class definitions, we observe the following:

- Both classes have two data elements *x* and *y*. In the class **Point** these elements describe the position of the point, in the case of class **Circle** they describe the circle's center. Thus, *x* and *y* have the same meaning in both classes: They describe the position of their associated object by defining a point.
- Both classes offer the same set of methods to get and set the value of the two data elements *x* and *y*.
- Class **Circle** ``adds'' a new data element ***radius*** and corresponding access methods.

Thus knowing the properties of class **Point** we can describe a circle as a point plus a radius and methods to access it. Thus, a circle is ``**A-KIND-OF**'' point. However, a circle is somewhat more ``specialized''as depicted in Figure 1.

Fig. 1 Illustration of ``A-KIND-OF'' relationship



1.7 Part-Of Relationship

We sometimes need to be able to build objects by combining them out of others. For example, a **Aeroplane** consists of engine, tyres, brakes, seats and others. This can be modelled as structures of classes using a vaious types of data put together.

Following Table provides a snap shot of some commonly obeserved relationships.

Verbatime		Relashionship
Animal is a human	Human is a Animal	Is-A
Car has place to sit	Truck has a place for hauling	Has-A
Fire is due to smoke	Smoke is due to Fire	Has-A
Passangers use Train	Train has Passangers	Has- a / Use - A
An employee is a Manager	Manager is an employee	MulitpleHas-A, Is-A
A building is a structre has many rooms	Building has many rooms and it's a structure	Part-of, has-A
Cas consists of parts(engines, tyres, seats, brakes, etc.)	Car/Truck/Bike is a Vehicle Aeroplane is an air vehicla	Parts-Of, Has-A

II. MULTIPLE INHERITANCE AND INTERFACES

One important object-oriented paradigm is multiple inheritance and interfaces. Multiple inheritance does **not** mean that multiple subclasses share the same superclass. It also does **not** mean that a subclass can inherit from a class which itself is a subclass of another class.

Multiple inheritance means that one subclass can have **more than one** superclass. This enables the subclass to inherit properties of more than one superclass and to ``merge'' their properties.

A *String* class supported by C / C++ / Java / C# , allows convenient handling of text. For example, we have a method to append text. We can use this class to add text to the possible drawing objects. It would be nice to also use already existing routines such as move()or ***draw()*** to move the text around. Thus it makes sense to let a drawable text have a point which defines its location within the drawing area. Hence we derive a new class *DrawableString* which inherits properties from *Point* and *String* as depicted in Figure 2.

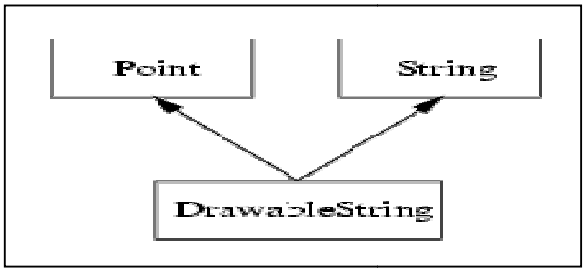


Figure 2 Derive a drawable string which inherits properties of *Point* and *String*.

We can write this by simply separating the multiple superclasses by comma:

```
class DrawableString inherits  from Point, String {
attributes:
    /* All inherited from superclasses */
methods:
    /* All inherited from superclasses */
  }
```

We can use objects of class *DrawableString* like both points and strings. Because a drawablestring is-a point we can move them around

```
DrawableString dstring
...
move(dstring, 10)
...
```

Since it is a **string**, we can append other text to them:

```
dstring.append(" Hyderabadi Beeryani ka Jabab Naahi ...")
```

In essensce ,if class Hyderabadi inherits from more than one class, ie. Hyderabadi - Tanglish (A) inherits from Hindi (L1), Telugu (L2), English(L3) and Urdu (L4) ... , Ln, we say it as multiple inheritance. Generalzing this, if A inherits from B1, B2…Bn, this may introduce naming conflicts in A if at least two of its super classes define properties with the same name.

The above definition introduce *naming conflicts* which occur if more than one superclass of a subclass use the same name for either attributes or methods. For an example, assuming, that class *String* defines a method *setX()* which sets the string to a sequence of ``X'' characters. The question arises, what should be inherited by *DrawableString*? The *Point*, *String* version or none of them?

These conflicts can be solved in at least two ways:

- The order in which the superclasses are provided define which property will be accessible by the conflict causing name. Others will be ``hidden''.
- The subclass must resolve the conflict by providing a property with the name and by defining how to use the ones from its superclasses.

The first solution is not very convenient as it introduces implicit consequences depending on the order in which classes inherit from each other. For the second case, subclasses must explicitly redefine properties which are involved in a naming conflict. A special type of naming conflict is introduced if a class *D* multiply inherits from superclasses *B* and *C* which themselves are derived from one superclass *A*, called as Dimond problem, as shown in Figure 3.

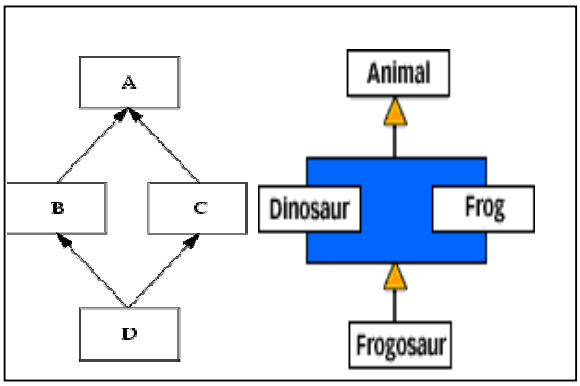


Figure 3. A name conflict introduced by a shared superclass

The question arises what properties class **D** actually inherits from its superclasses **B** and **C**. Some existing programming languages solve this special inheritance graph by deriving **D** with

- the properties of **A** plus
- the properties of **B** and **C** **without** the properties they have inherited from **A**.

Consequently, **D** cannot introduce naming conflicts with names of class **A**. However, if **B** and **C** add properties with the same name, **D** runs into a naming conflict. Another possible solution is, that **D** inherits from both inheritance paths. In this solution, **D** owns **two** copies of the properties of **A**: one is inherited by **B** and one by **C**.

Although multiple inheritance is a powerful object-oriented mechanism the problems introduced with naming conflicts have lead several authors to ``**doom**'' it. As the result of multiple inheritance can always be achieved by using (simple) inheritance some object-oriented languages even don't allow its use, especially Java. However, carefully used, under some conditions multiple inheritance provides an efficient and elegant way of formulating things

2.1 Multiple Interfaces [4][5][6]

Java language was designed without multiple inheritance. While it may seem as a design flaw, it is actually true that the overall design of Java supports the solution of problems commonly solved with multiple inheritance in other ways. In particular, the singly rooted hierarchy (with **Object** as the ultimate ancestor of all classes) and Java interfaces solves most problems that are commonly solved using multiple inheritance in C++

2.11 Mixin Inheritance

In mixin inheritance, one class is specifically designed to be used as one of the classes in a multiple inheritance scheme. We say that it provides some functionality that is "mixed in" to some other class that wants this functionality. Another way of looking at mixin inheritance is that a mixin class is given a new parent class so that the mixin seems to extend the other class. In some projects it is necessary to rely on common services that must be provided by several classes. Mixin inheritance is one way to centralize the development of these services. To provide for mixin inheritance we will need to define two interfaces as well as at least one class that provides the service: the Mixin class. In some situations, one of these interfaces is empty and may be omitted

2.12 The Interfaces

The first interface (always required) defines what the mixin class will provide for services. It defines one or more methods that will be implemented by the mixin class. We will take a simple and abstract example here. The class will be called (abstractly) **MusicProvides** to emphasize that it defines what any compatible mixin must provide. We will also assume that the only service provided is a void function and we will give it the abstract name play. In practice, however, there may be any number of methods defined and they may have any signatures.

```
interface MusicProvides
{    void play();
}
```

One special feature of mixin inheritance that is not usually present in the general multiple inheritance case is that the mixin class may require services of the class into which it is mixed. That is, in order to provide the "play" service, the mixin may need to get some information from the other class. We define this with another interface and give it the abstract name ***MusicRequires*** to indicate that it requires one or more services from the class into which it is mixed.

Supposing that the compatible mixins require the other class provides a method ***getValue()*** that returns an **int**.

```
interface MusicRequires
{    int getValue();
}
```

In general, MusicRequires will have a more appropriate name and will have one or more methods of arbitrary signature. However, any class into which we mix our mixin must provide services with the given names and signatures, though it need not explicitly implement the **MusicRequires** interface. If **MusicRequires** is empty it may be omitted.

2.13 The Mixin

The mixin class itself will implement the **MusicProvides** interface. It will also be created by passing its constructor an argument that implements **MusicRequires**. A simple example called (again abstractly) **Mixin** shows this :

```
class Mixin implements MusicProvides
{    public Mixin(MusicRequires parent) {
this.parent = parent; }

    public void play() { System.out.println("The
value is: " + parent.getValue()); }

    private final MusicRequires parent;
}
```

When a new **Mixin** is created it knows about an **MusicRequires** object. It can then query this object using the services defined in **MusicRequires** in order to provide its own services. We have called this object parent to emphasize that the intent is to simulate giving the **Mixin** class a new parent class. If **MusicRequires** is empty, no object need be passed into the constructor, note that in general, the **Mixin** constructor may require other parameters as well.

2.14 The Class Used as the Base

Now suppose that we wish to mix this class into another class. This class must have all of the methods required of the MusicRequires interface, but it need not implement this interface. It will probably have other methods as well. Here is an example:

```
Class Parent
{    public P(int value ) { this.val = value; }
```

```
    public int getValue() { return this.val; }
    public toString() { return "" + this.val; }
    private int val;
}
```

2.15 Result of Mixing

Now, to actually mix the two classes together we first build a new class that extends Parent and implements both of our interfaces

```
class Child extends Parent implements
MusicRequires, MusicProvides
{ …. }
```

This class defines both the services of Parent and those of the mixin (**MusicProvides**). To implement the **Child** class we create a new **Mixin** object and save it. We will also delegate all messages defined in the **MusicProvides** interface to this object. When we create the **Mixin** object we need to pass it an object that implements **MusicRequires**, but this object does so as this new class implements the **MusicRequires** interface.

```
    public Child(int value)
    {       super(value);
            this.mixin = new Mixin(this);
    }
    public void play(){ mixin.play(); }
    private final MusicProvides mixin;
}
```

Note that the service named ***play***() is provided by the **Mixin** object as previously defined. It does not need to be redefined in the **Child** class and hence the Child class automatically implements **MusicRequires** since the inherited method fulfills the necessary contract defined by the **MusicRequires** interface. We do have to provide the simple delegation function for the method(s) of MusicProvides, however.

The key here is that we were able to design the Mixin class to be a mixin and so define the two required interfaces. Note that it is actually the services defined by MusicProvides that are mixed in, not, properly speaking, the Mixin class itself. This actually adds some flexibility since several classes might implement this interface and so be mixed in to other classes to provide this form of multiple inheritance.

III. WRAP UP (CONCLUSION)

In this paper, an attempt is made to make the abstract concepts of OO techniques in a prolific and understandable way by exploring various relashionships in varying and their suitability contexts. The adeptness of the underlying principles makes them appropriately fixing for a particlar paradigm. The vaious principles and domain analysis makes a lucid and conscise way to make their appliactions suitability.

BIBLIOGRAPHY

[1] Robert Lafore , Object Oriented Programing in C++, Pearson Education , 2nd Ed. 2011

[2] Herbert Schildt. Java Cpmplete Reference, 7th Ed. THM 2010

[3] Miller Page – Jones , Fundamentals of Object Orinetd Design in UM, Pearson Edu. 2nd Ed. 2008

[4] AlisdairWren, relashioships for Object Oriented Programming Languages: A technical Report 702, University of Cambridge, Computer Laboratory, 2007

[5] Stepfan Van Baelen, johan Lewi, eric Steegmans and Helena Van riel, an Entity Relstiosmship based Object – Oriented Specification Method, Katholieke Universitiet Leuven, Belgium

[6] James Rambaugh, relashionship Sementic Constructs in an Object Oriented Language, corporate research and Delopment, NY 2008

ACKNOWLEDGEMENT

We are thankful to Prof. J. Mahender Reddy (VC-IFHE) , Dr. M.Raja (Advisor, FST) Prof. Murthy Gutta (I/C Dean, FST) for their valuable support and inspirations. Special thanks to our family members for their continued support and endurance.

◈◈◈