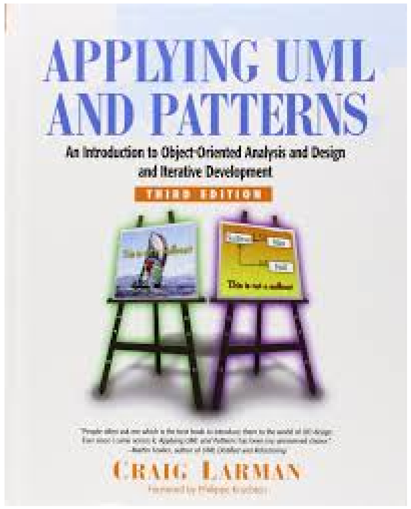
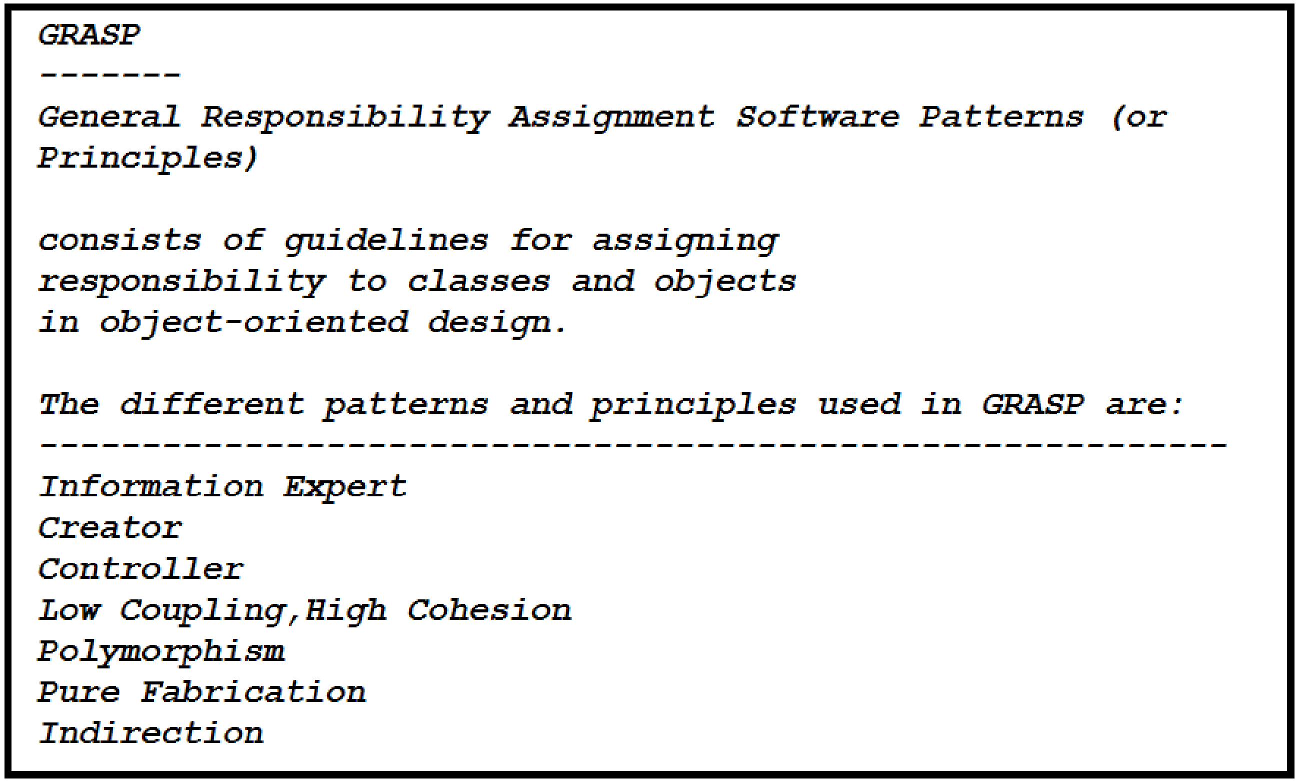
Designing Shopping Cart OOAD
The Expert, the Aggregate and the Value Object.

Rajeev Gupta
Java trainer
Rgupta.mtech@gmail.com


APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and Iterative Development
THIRD EDITION
CRAIG LARMAN

GRASP

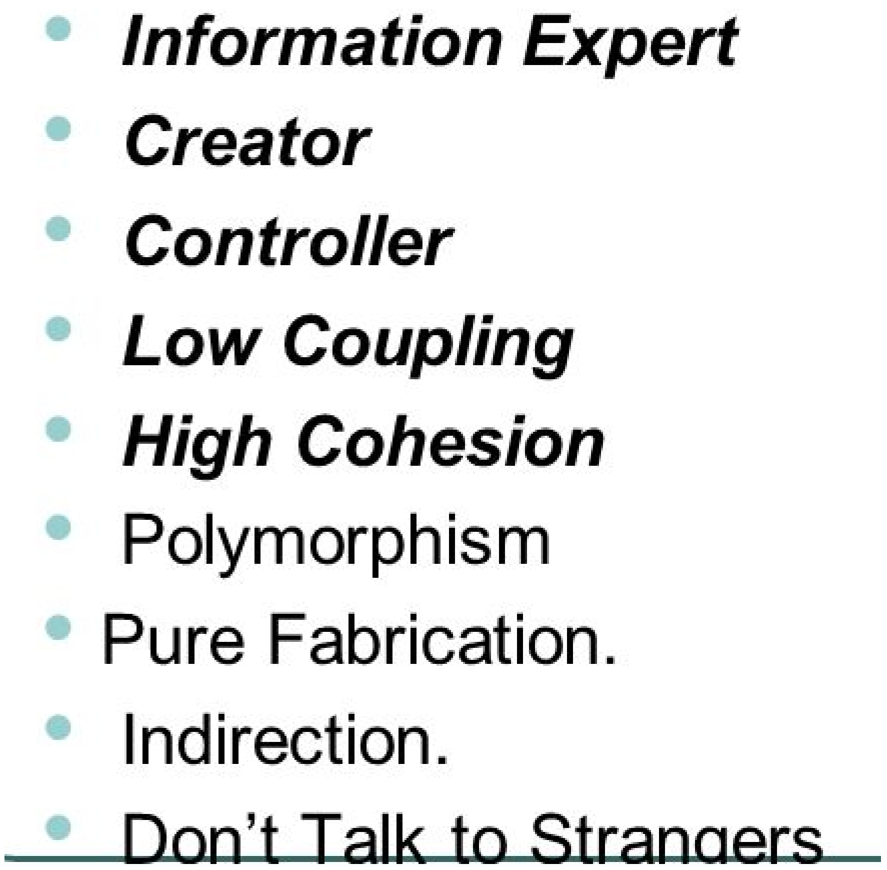


```
GRASP
-------
General Responsibility Assignment Software Patterns (or
Principles)

consists of guidelines for assigning
responsibility to classes and objects
in object-oriented design.

The different patterns and principles used in GRASP are:
------------------------------------------------------------
Information Expert
Creator
Controller
Low Coupling,High Cohesion
Polymorphism
Pure Fabrication
Indirection
```

9 GRASP Patterns



- ***Information Expert***
- ***Creator***
- ***Controller***
- ***Low Coupling***
- ***High Cohesion***
- Polymorphism
- Pure Fabrication.
- Indirection.
- Don’t Talk to Strangers

Using GRASP and Domain Driven Design patterns to solve business problems

We use folloing patterns to solve shopping cart problem

- The Information Expert pattern.
- The Aggregate Root pattern.
- The Value Object pattern.

We'll see how wrongly managing responsibilities and behaviors, while abusing getters and setters, can lead to a corrupted business domain, and how the patterns above will help you to correctly address these kind of problems.

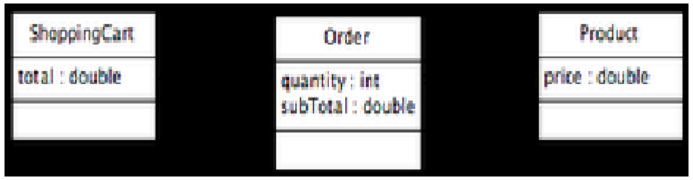
Business Case

A ShoppingCart can hold one or more Orders, and an Order refers just to a single Product.
Moreover, we have some simple business rules regarding these entities:

The ShoppingCart total is the sum of all Order sub-totals.

Each Order sub-total is the Product price times the ordered Product quantity.

The ShoppingCart total cannot exceed the value of EUR 10000.(**invariant)**



The business domain.

- Let's start consider a simple business domain
- We have a shopping cart holding a number of orders, each referring to a given product.
  let's restrict our domain model to three simple entities:
  - ShoppingCart,
  - Order and
  - Product.

The business domain.

- A ShoppingCart can hold one or more Orders, and an Order refers just to a single Product.
  Moreover, we have some simple business rules regarding these entities:
  1. The ShoppingCart total is the sum of all Order sub-totals.
  2. Each Order sub-total is the Product price times the ordered Product quantity.

  3. The ShoppingCart total cannot exceed the value of Rs: 10000.

```
public class Product
 {

     private String name;

     // ...

     private double price;

     // ...

     public double
     getPrice()
        {
     return this.price;
     }

     public void
     setPrice(double price)
         {
     this.price = price;
     }
}
```

```
public class Order
{


       private Product product;
      private int quantity;

      public Product getProduct() {
            return this.product;
        }

      public void
setProduct(Product p) {
         this.product = p;
          }

      public int getQuantity() {
        return this.quantity;
     }

    public void setQuantity(int q) {
          this.quantity = q;
    }
}
```

```
public class ShoppingCart
{

     private List orders = new
ArrayList();

      public void addOrder(Order o)
     {
              this.orders.add(o);
      }

   public List getOrders()
     {
     return this.orders;
    }
}
```

As you can see, all classes have the right properties and dependencies:

1. A Product has a price.
2. An Order refers a Product and has a product quantity.
3. The ShoppingCart has more orders.
4. The ShoppingCart total can be calculated by iterating all Orders and summing up each Product price times the ordered product quantity.

Problem with previous design

- Previous design (and implementation) is seriously flawed: There's no clear assignment of responsibilities and behaviors.
  - Hence, it is full of getters and setters.
    - Hence, there's **no encapsulation**.
      - Hence, all business logic is **external**.
        - » Hence, **it is easy to corrupt domain state and violate business rules**.

What is undone...

```
Product p1 = new Product();
Product p2 = new Product();

p1.setPrice(1000);
p2.setPrice(2000);

Order o1 = new Order();
Order o2 = new Order();

o1.setProduct(p1);
o1.setQuantity(5);

o2.setProduct(p2);
o2.setQuantity(5);

ShoppingCart cart = new
ShoppingCart();

cart.addOrder(o1);
cart.addOrder(o2);
```

But the ShoppingCart total is now Rs. 10000!

That's because the business logic is computed outside of the object that holds the information; so let's solve this problem by introducing the ***Information Expert pattern*.**

Refactoring toward the Expert.

- The Information Expert pattern is part of the **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns (**GRASP**), and states the following:

Assign a responsibility to the information expert: the class that has the information necessary to fulfill the responsibility.

What responsibilities to relocate?

- We have two responsibilities to relocate:
  - The shopping cart total computation: here, **the information expert is the ShoppingCart class**.
  - The order sub-total computation: here, **the information expert is the Order class**.

- By relocating the two responsibilities we'll improve our encapsulation and make our implementation more robust, because the ShoppingCart will be able to check that the total doesn't exceed EUR 10000.

```
public class Order {

    private Product
    product;
    private int quantity;

    public void
    setProductWithQuanti
    ty(Product p, int q) {
    this.product = p;
    this.quantity = q;
    }

    public double
    computeSubTotal() {
    return
    this.product.getPrice(
    ) * this.quantity;
    }
    }
```

As you can see, the ShoppingCart *computeTotal()* method is now able to check against the maximum total, and throw an exception if there's something wrong, avoiding domain state corruption.

```
public class ShoppingCart {

            private List orders = new ArrayList();

            public void addOrder(Order o) {
            this.orders.add(o);
            }

            public double computeTotal()
                {
                        double total = 0;
                        Iterator it =
this.orders.iterator();
                        while (it.hasNext())
                         {
                        Order current = (Order)
it.next();
                        total +=
current.computeSubTotal();
                                    if (total > MAX_TOTAL)
                                     {
                                    throw new SomeException();
                                    }
                        }
            return total;
            }
}
```

Problem still persist!!!

```
Product p1 = new Product();
Product p2 = new Product();

p1.setPrice(1000);
p2.setPrice(2000);

Order o1 = new Order();
Order o2 = new Order();

o1.setProductWithQuantity(p1, 2);
o2.setProductWithQuantity(p2, 4);

ShoppingCart cart = new
ShoppingCart();

cart.addOrder(o1);
cart.addOrder(o2);

cart.computeTotal();

// !!!!!!!!! DANGER !!!!!!!!!!
o1.setProductWithQuantity(p1, 3);
// !!!!!!!!! DANGER !!!!!!!!!!
```

The problem is that we can always change the Order product and quantity, changing so the shopping cart total without preventing it to enter in an invalid state!

Where's the **real** problem?
How to solve it?

Refactoring toward the Aggregate and the Value Object.

The real problem is…

- The real problem is that we are interested in keeping the ShoppingCart state **always correct**, that is, in keeping its **invariants**: however in the current design and implementation we are not able to control everything happens inside the ShoppingCart, because we can directly modify its Orders without going through it!

  The solution is to apply the Aggregate and Value Object patterns, part of the Domain Driven Design.

Aggregate

- An *Aggregate* is a set of related objects whose invariants must always be kept consistent, and an *Aggregate Root* is an object that acts like the main access point into the aggregate;

- all access must go through the root, and objects external to the aggregate cannot keep references to objects contained into the aggregate: they can keep a reference only to the aggregate root.

Value Object

- A *Value Object* is an object that has no identity and is immutable: it is equal to another value object of the same type if its properties are equal too, and must be discarded if its properties need to change.

  How to turn this theory into practice, applying it to our domain?

Applying Aggregate

- First, **our ShoppingCart and Order are part of an aggregate**.
  It's easy: the "EUR 10000" invariant involves both objects, so it must be kept consistent across the two.

- Moreover, Orders make sense only if related to a ShoppingCart, so no one should access an Order without first going through a ShoppingCart: this means that **the ShoppingCart is the root of the aggregate**.

Applying Value Object

- Second, **the Order is a Value Object**: it doesn't make sense to create an order and change its related product and quantity during its life cycle;

- an order always refers to the same product with the same quantity, and if something needs to be changed, it must be discarded and a new one

```
public class Order {

    private Product
    product;
    private int quantity;
    private double
    subTotal;

    public Order(Product
    p, int q) {
    this.product = p;
    this.quantity = q;
    this.subTotal =
    this.product.getPrice(
    ) * this.quantity;
    }

    public double
    computeSubTotal() {
    return this.subTotal;
    }
}
```

The Order class is now immutable and part of an aggregate together with the ShoppingCart. The ShoppingCart is now the aggregate root, and every change to its Orders must go through it.

```
public class ShoppingCart {

private List orders = new ArrayList();

public void addOrder(Order o) {
this.orders.add(o);
}

public boolean removeOrder(Order o) {
return this.orders.remove(o);
}

public double computeTotal() {
double total = 0;
Iterator it = this.orders.iterator();
while (it.hasNext()) {
Order current = (Order) it.next();
total += current.computeSubTotal();
}
return total;
}
}
```

Now, **you have no way of corrupting your domain**.

Remember: *the Expert, the Aggregate and the Value Object*.