Cohesion & Coupling Examples M27CDE

Coupling

Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled…that's a good thing.

If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…not a good thing.

In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

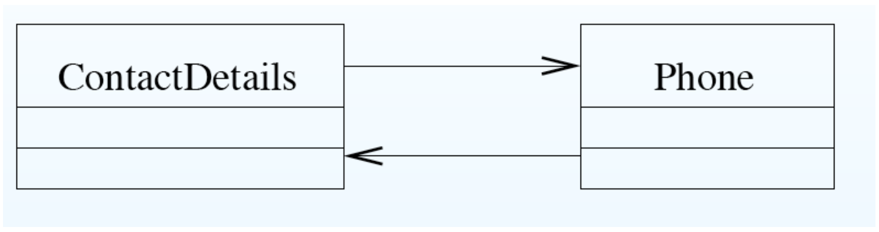
Using this second scenario, imagine what happens when class B is enhanced. It's quite possible that the developer enhancing class B has no knowledge of class A, why would she?

Class B's developer ought to feel that any enhancements that don't break the class's interface should be safe, so she might change some non-interface part of the class, which then causes class A to break.

At the far end of the coupling spectrum is the horrible situation in which class A knows non-API stuff about class B, and class B knows non-API stuff about class A…this is REALLY BAD. If either class is ever changed, there's a chance that the other class will break. An example of tight coupling, which has been enabled by poor encapsulation:

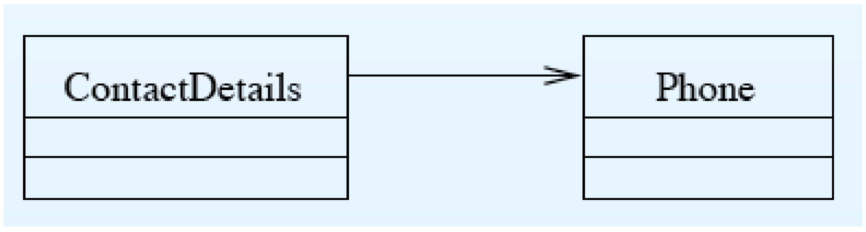
Example 1: High Coupling

```
public class ContactDetails {
   private Phone phone;
   private String street;
   public String city;

   public ContactDetails(Phone p, String s, String c) {
      phone = p; street = s; city = c;
   }

   public void print() {
      System.out.println( street);
      System.out.println( city);

   }

}


public class Phone {
   private public area;
   private public number;

   public Phone(int a, int n, ContactDetails c) {
      area = a; number = n;
      validateArea( area, c.city);
  }

}
```



Example 1: Low Coupling

```
public class ContactDetails {
   private Phone phone;
   private String street;
   private String city;

   public ContactDetails(int a, int n, String s, String c) {
      validateArea(a, c);
      phone = new Phone(a,n);
      street = s; city = c;
   }

   public void print() {
      System.out.println(street);

      System.out.println(city);
      phone.printPhone();
   }

   public String getCity() {
      return city;
   }
}


public class Phone {
   private int area;
   private int number;
   public Phone(int a, int n) {
      area = a; number = n;
  }

  public printPhone(){

      System.out.println(area+” ”+number);

  }
}
```



Example 2: High / Low Coupling

```
class Traveler
{
    Car c=new Car();
    void startJourney()
    {
       c.move();
    }
}

class Car
{
  void move()
  {
     // logic...
  }
}
```

In the above example, Traveler object is depends on car object. So traveler class creating an object of Car class inside it.

If the other class object is created in the dependent class (like Car class object in Traveler class), there exist tight coupling, If method in car object is changed then we need to do the changes in the Traveler class too so its the tight coupling between Traveler and Car class objects

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
      this.v = v;
    }

    void startJourney()
    {
       v.move();
    }
}


Interface Vehicle
{
   void move();
}


class Car implements Vehicle
{
    public void move()
    {
         // logic
    }
}


class Bike implements Vehicle
{
    public void move()
    {
         // logic
    }
}
```

In above example, we can use an interface to define common Vehicle objects.

We can then inject either Car, Bike or any other object implementing Vehicle into the Traveler by calling setter method,

So if Car object is replaced with Bike then no changes are required in Traveler class, this means there is loose coupling between Traveler and Vehicle object.

Cohesion

Cohesion is all about how a single class is designed. The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose.

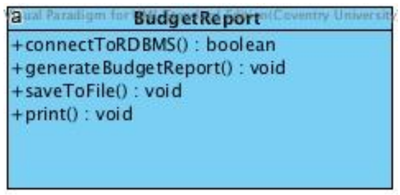
Keep in mind that cohesion is a subjective concept. The more focused a class is, the higher its cohesiveness—a good thing. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion.

Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

Let's take a look at a pseudo-code example:

Example: Low Cohesion

```
class BudgetReport {

   connectToRDBMS(){ }
   void generateBudgetReport() { }
   void saveToFile() { }
   void print() { }

}
```



Now imagine your manager comes along and says, "Hey you know that accounting application we're working on? The clients just decided that they're also going to want to generate a revenue projection report, oh and they want to do some inventory reporting also. They do like our reporting features however, so make sure that all of these reports will let them choose a database, choose a printer, and save generated reports to data files..."

Rather than putting all the printing code into one report class, we probably would have been better off with the following design right from the start:

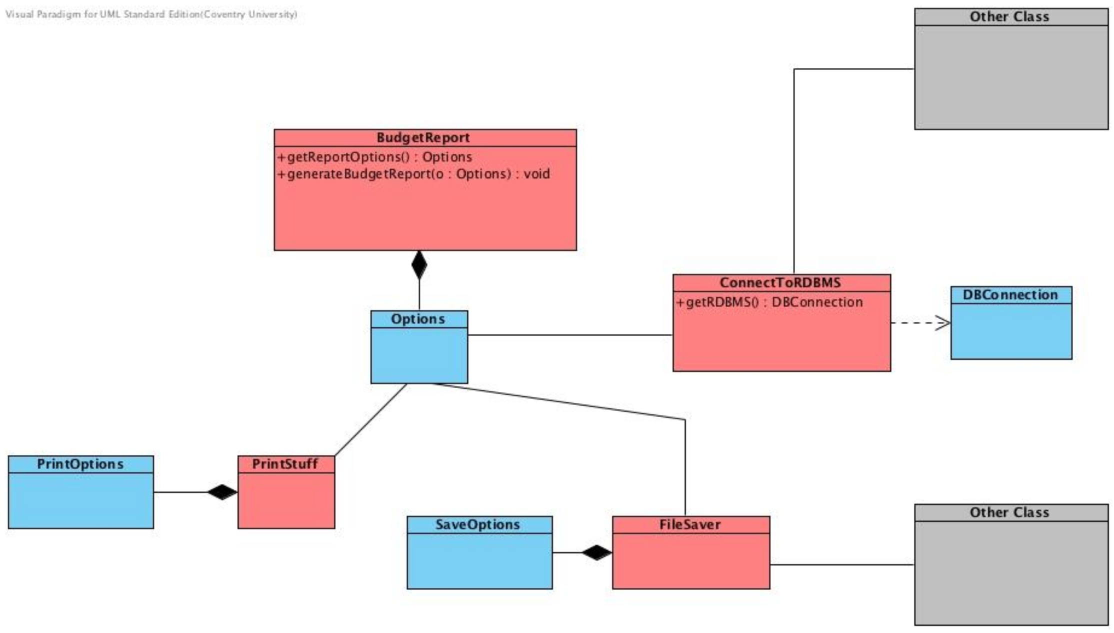
Example: High Cohesion

```
class BudgetReport {
   Options getReportingOptions() { }
   void generateBudgetReport(Options o) { }
}

class ConnectToRDBMS {
   DBconnection getRDBMS() { }
}


class PrintStuff {
   PrintOptions getPrintOptions() { }
}

class FileSaver {
   SaveOptions getFileSaveOptions() { }
}
```

This design is much more cohesive.

Instead of one class that does everything, we've broken the system into four main classes, each with a very specific, or cohesive, role.

Because we've built these specialized, reusable classes, it'll be much easier to write a new report, since we've already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report.



Visual Paradigm for UML Standard Edition(Coventry University)
Other Class
BudgetReport
+getReportOptions() : Options
+generateBudgetReport(o : Options) : void
ConnectToRDBMS
+getRDBMS() : DBConnection
DBConnection
Options
PrintOptions
PrintStuff
SaveOptions
FileSaver
Other Class

Useful Links on Coupling and Cohesion:

http://msdn.microsoft.com/en-us/magazine/cc947917.aspx

http://edn.embarcadero.com/article/30372

Adapted from: http://www.primeinspiration.com/more/programming/plguides/914-coupling-and-cohesion.html