Software Design: Coupling and Cohesion

Acknowledgement

- Some of the contents are adapted from material by Manzil-e-Maqsood

- Software Engineering – A practitioner’s approach by Roger S. Pressman and Bruce. R. Maxim

- Software Engineering by Ian Sommerville

Result of Feedback

- Things to keep:
  - Interactive class
  - Quizzes
  - Slides are clear
- Some feedback I'll follow up on immediately:
  - More Tophat questions for participation
  - More clarification about project and expectations
  - Quick marking
  - Number of assignments
  - In class time to work on group projects
  - Less participation from project members
  - Working on projects during tutorials
  - Comprehensibility of the documents (quizzes, assignments etc.)
- Things I can't change:
  - Increase weight-age of certain components
  - Removing components from the course
  - One real question with one real answer

Recap

- A maintainable design is the best design
- Advantages:
  - Cost effective
  - Easy to change

Recap

- Modularity
  - Allows separation of aspects to be implemented for the system (separation of concerns)

- Cohesion
  - Intra-component relationship
  - Strength of the module within itself

Recap…

- Coupling
  - Inter-component relationship
  - Highly coupled components have strong inter-relations

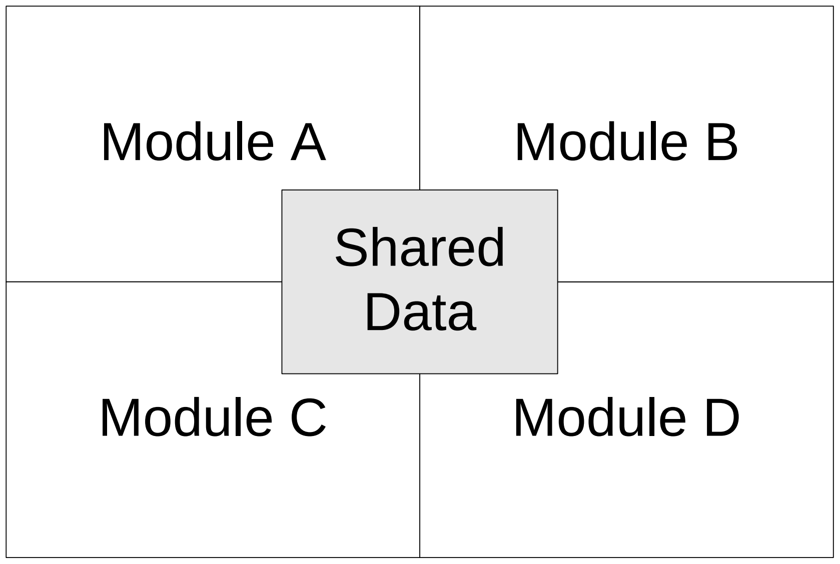
High Cohesion;

Low Coupling

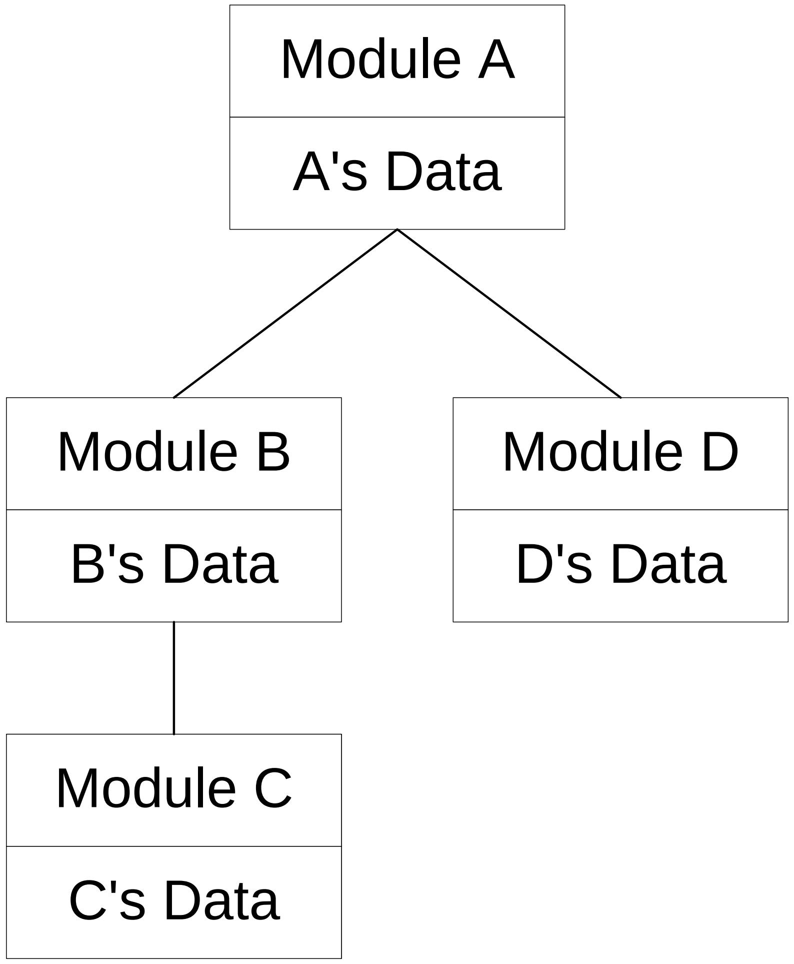
More about Coupling

- High coupling is difficult to test, reuse, and understand separately.
- Loosely coupled modules are independent or almost independent.

Examples



High Coupling



Low Coupling

The modules that interact with each other through message passing have low coupling while those who interact with each other through variables that maintain information about the state of the module have high coupling

Coupling- An Example

```
class vector {

                public float x;
                public float y;
                public vector(float x, float y){//some code here}
                public float getX(){//some code here;}
                public float getY(){//some code here;}
                public float getMagnitude(){//some code here;}
                public float getAngle(){//some code here;}
}
```

```
float myDotProduct1(vector a, vector b)
{
                float temp1 = a.getX() * b.getX();
                float temp2 = a.getY() * b.getY();
                return temp1 + temp2;
}
```

```
float myDotProduct2(vector a, vector b)
{
                float temp1 = a.x * b.x;
                float temp2 = a.y * b.y;
                return temp1 + temp2;
}
```

```
class vector {

        public float magnitude;
        public float angle;
        public vector(float x, float y){//some code here}
        public float getX(){//some code here;}
        public float getY(){//some code here;}
        public float getMagnitude(){//some code here;}
        public float getAngle(){//some code here;}
```

Coupling increases when you access data members of a class

Change is spread out in highly coupled classes

Avoid coupling:

- To reduce coupling you should therefore *encapsulate* all instance variables
  - declare them private
  - and provide get and set methods
- A worse form of coupling occurs when you directly modify an instance variable.

Cohesion- Example

```
class order {
   public:
            int getOrderID();
            date getOrderDate();
            float getTotalPrice();
            int getCustometId();
            string getCustomerName();
            string getCustometAddress();
            int getCustometPhone();
            void setOrderID(int oId);
            void setOrderDate(date oDate);
            void setTotalPrice(float tPrice);
            void setCustometId(int cId);
            void setCustomerName(string cName);
            void setCustometAddress(string cAddress);
            void setCustometPhone(int cPhone);
            void setCustomerFax(int cFax)
   private:
            int orderId;
            date orderDate;
            float totalPrice;
            item lineItems[20];
            int customerId;
            string customerName;
            int customerPhone;
            int customerFax;
};
```

Cohesion- Example …

```
class order {
    public:
            int getOrderID();
            date getOrderDate();
            float getTotalPrice();
            int getCustometId();
            void setOrderID(int oId);
            void setOrderDate(date oDate);
            void setTotalPrice(float tPrice);
            void setCustometId(int cId);
            void addLineItem(item anItem);
    private:
            int orderId;
            date orderDate;
            float totalPrice;
            item lineItems[20];
            int customerId;
};
```

Cohesion Example …

```
class customer {
   public:
            int getCustometId();
            string getCustomerName();
            string getCustometAddress();
            int getCustometPhone();
            int getCustomerFax();
            void setCustometId(int cId);
            void setCustomerName(string cName);
            svoid setCustometAddress(string cAddress);
            void setCustometPhone(int cPhone);
            void setCustomerFax(int cFax);
   private:
            int customerId;
            string customerName;
            int customerPhone;
            int customerFax;
};
```

Design Heuristics for Classes

- Coupling and cohesion
  - Keep related data and behavior in one place
  - Spin-off non-related information into another class

Temporal Cohesion

- *Operations that are performed during the same phase of the execution* of the program are kept together, and everything else is kept out
  - For example, placing together the code used during system start-up or initialization.

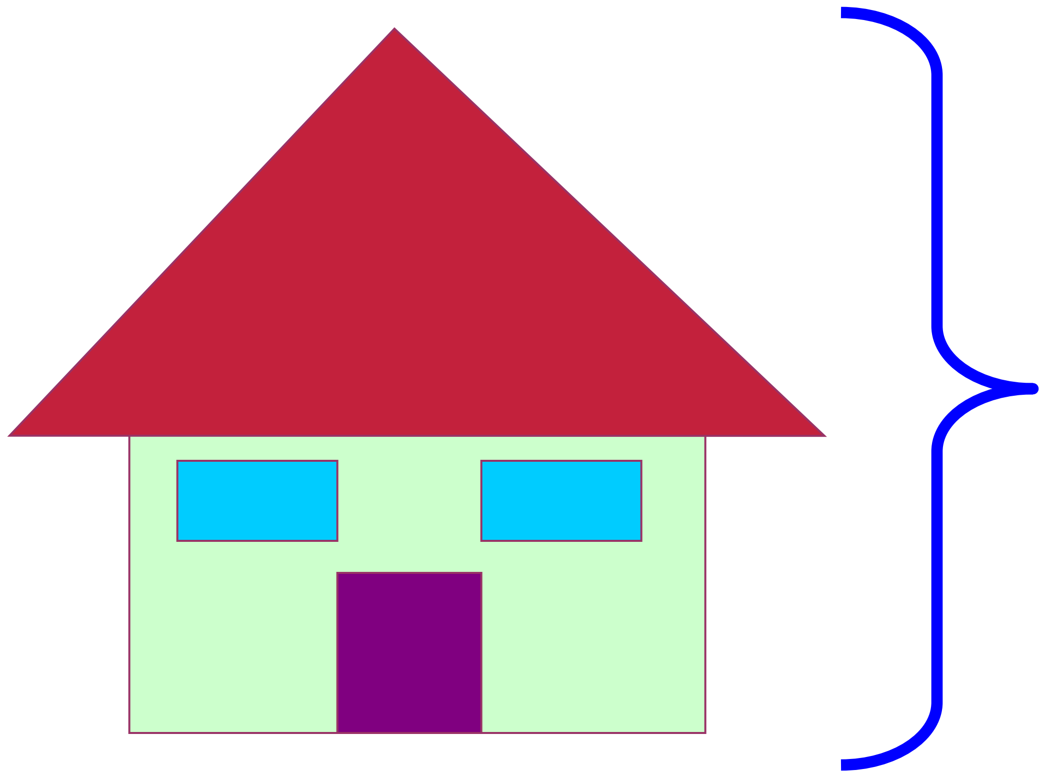
Utility cohesion

- When *related utilities which cannot be logically placed in other cohesive units* are kept together
  - A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
  - For example, the **java.lang.Math** class.

Abstraction
&
Encapsulation (Information Hiding)

Abstraction

- Abstraction is a means of reducing complexity by handling different details at different levels



House is an abstraction for a collection of doors, windows, walls and roof

Town is an abstraction for a collection of houses

Encapsulation/ Information Hiding

- Hides internal complexity
- Provides abstraction to the outside world
  - If you provide abstraction, you manage complexity
- Mother concept of Object Orientation
  - Encourages high cohesion
    - Data and operations regarding one aspect is placed at one place, i.e., all info is encapsulated at one place
  - Discourages high coupling
    - If data is spread out, working on it is difficult

Next

- Software Architecture