# Software Design

## Chapter 12 Pressman

# 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

# Today

- Simple vs. complex systems
- Managing complexity of a system
- Function oriented vs. object oriented design
- Characteristics of a good design

# Simple Systems

- Easily analyzed, designed, coded and debugged by a single person
- Components are small and manageable
- The system is not evolved

# Complex Systems

- Requires team effort
- Evolve from smaller, simpler systems
- Complexity increases through evolution
- Complexity becomes difficult to manage

# How to manage complexity of large systems?

- A complex system may be decomposed into smaller pieces of lesser complexity, called modules/components

# How to manage complexity of large systems?...

- Apply principles of:
  - Separation of concerns
    - Allows us to deal with different individual aspects of a problem by considering these aspects in isolation and independent of each other
    - E.g., an purchase inventory sub-system and a customer-relationship management sub-system can be treated independent of each other
  - Modularity
    - The classic divide-and-conquer philosophy
    - Allows the designer to apply the principle of separation of concern on individual modules
  - Abstraction
    - Information hiding

# What is Software Design?

- **A roadmap or a bridge from requirements towards implementation**
- Software requirements– **Domain of What?**
- Design– **Domain of How?**
  - How to realize the system based on software requirements?

# What is included in a software design?

- Modeling of components and entities
- Interfaces between different components of the system
- Interfaces to the outside world

# Software Design Strategies

- Function Oriented / Structured Design

- Object Oriented Design

# Function Oriented / Structured Design

- The entire system is abstracted as a function that provides the desired functionality.
- A function is divided into other functions until it can be easily implemented.
- The data (also called system state) maintained by the system, is centralized and is shared among these functions.

# Object Oriented Design

- System is decomposed into a set of objects that cooperate and coordinate with each other to implement the desired functionality.

- The data maintained by the system (also called system state) is decentralized and each object is responsible for maintaining its own state.

- OO approach generates design that is more maintainable

# Quality of a Design

- If you have more design options, how would you evaluate them?
  - Is that design a good one which results in an efficient software?
  - Is that design a good one which results in a software written in minimum lines of code?
- **The answer:** Both could be right depending upon the requirement

**BUT**

A very important and the most desired quality parameter related to software design is **Maintainability**

# What is a maintainable design?

- In which bringing about any change is easy
- Change could be
  - Fixing a bug
  - Enhancing the software
  - Adapting the software to a new environment
- If change is manageable, overall cost decreases and overall profit margins increase

# Important concepts related to software maintainability

- Cohesion
- Coupling

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

# Example of Coupling

- Example 1
  - Weight bearing wall and the roof
  - if you break the wall, roof will fall
  - Roof is highly dependent on wall
- Example 2
  - Electrical wiring of a room done in series circuits
  - If one thing stops working, everything will stop working
  - High coupling; not recommended

# Example of Cohesion

- Example 1
  - Calculate freight
  - Contains information only relevant to calculating freight
  - High cohesion ; recommended
- Example 2
  - Date object
  - Data members = day, month, year, red, apple
  - ‘Red’ and ‘apple’ data members make the date object less cohesive; is irrelevant to the logic of the object date

# 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;
};
```

# Why low coupling and high cohesion?

- Systems with loosely coupled components are highly independent of each other and are easier to fix and change.

- **The changes are localized** and not spread apart; decreases risk of ripple effects

- Objects with high cohesion are easily reused.

# Cohesion: 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.

# Cohesion: 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.

# Summary

- Simple vs. complex systems
- Managing complexity of a system
- Function oriented vs. object oriented design
- Characteristics of a good design
- Types of cohesion
  - Temporal and Utility Cohesion

# Next Lecture

- Software Architecture