1. What is Agile Design?

Ans: If agility is about building software in tiny increments, how can you ever design the software?

- How can you take the time to ensure that the software has a good structure that is flexible, maintainable, and reusable?
- If you build in tiny increments, aren't you really setting the stage for lots of scrap and rework in the name of refactoring?
- Aren't you going to miss the big picture?
- The big picture evolves along with the software. With each iteration, the team improves the design of the system so that it is as good as it can be for the system as it is now.

2. Mention symptoms of poor design in brief.

Ans: Symptoms of Poor Design:

- **Rigidity**: The design is hard to change.
- **Fragility**: The design is easy to break
- **Immobility**: The design is hard to reuse.
- **Viscosity**: It is hard to do the right thing.
- **Needless Complexity**: Overdesign.
- **Needless Repetition**: Mouse abuse.
- **Opacity**: Disorganized expression.

3. Mention the odors of Rotting Software's.

Ans: Design Smells -The Odors of Rotting Software:

- **Rigidity**: The system is hard to change because every change force many other changes to other parts of the system.
- **Fragility**: Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
- **Immobility**: It is hard to disentangle the system into components that can be reused in other systems.
- **Viscosity**: Doing things right is harder than doing things wrong.
- **Needless Complexity**: The design contains infrastructure that adds no direct benefit.
- **Needless Repetition**: The design contains repeating structures that could be unified under a single abstraction.
- **Opacity**: It is hard to read and understand. It does not express its intent well

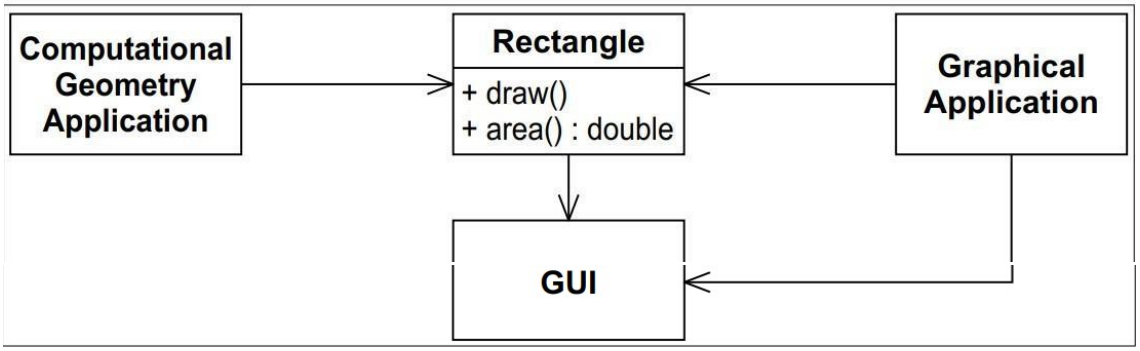
4. Explain mention below principle with diagrams and appropriate example.

Ans: Object-oriented design that help developers eliminate design smells and build the best designs for the current set of features.

- The principles are as follows:
    1. SRP: The Single Responsibility Principle
    2. OCP: The Open-Closed Principle.
    3. LSP: The Liskov Substitution Principle.
    4. ISP: The Interface Segregation Principle.
    5. DIP: The Dependency Inversion Principle.

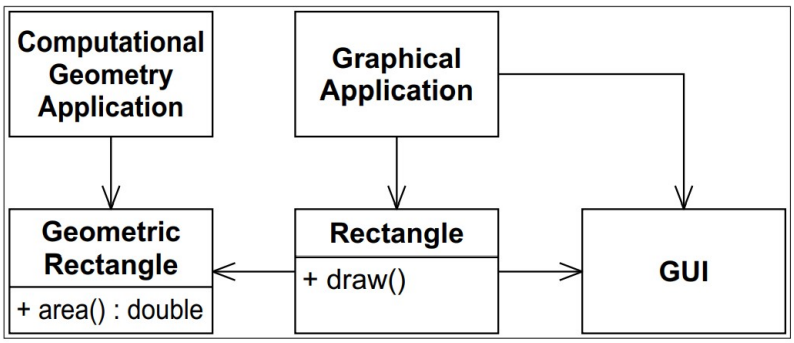
SRP: The Single Responsibility Principle:

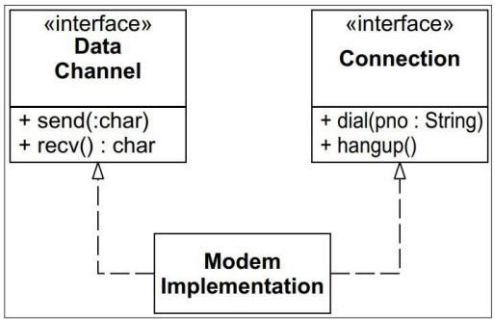
Each class should only be responsible for a single cohesive responsibility. Each responsibility is an axis of change. When the demand changes, the change will be reflected as the change in the responsibility of the class. A class should have only one reason for its change. If a class assumes more than one responsibility, then there will be multiple reasons for its change, which is equivalent to coupling these responsibilities together. Designs that violate SRP can usually be reconstructed using facade mode or agent mode to separate business responsibilities.



More than one responsibility:

- Two different applications use the Rectangle class.
    - One application does computational geometry. It uses Rectangle to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen.
    - The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen
- A better design is to separate the two responsibilities into two completely different classes as shown above.
- This design moves the computational portions of Rectangle into the Geometric Rectangle class.
- Now changes made to the way rectangles are rendered cannot affect the Computational Geometry Application.





Responsibility:

We define a responsibility to be "a reason for change." If you can think of more than one motive for changing a class, then that class has more than one responsibility.

interface Modem {

public void dial(String pno); public void hangup();

public void send(char c); public char recv();

}

OCP: The Open–Closed Principle:

- Software entities (classes, packages, modules, etc.) should be extensible, but unmodifiable, that is, they are open for expansion and closed for modification. When expanding the behavior of an entity, there is no need to change the source code or binary code, that is The Open-Closed Principle. The key to the principle of opening and closing is abstract design. It is impossible for complete OCP. The most likely and most frequent changes should be abstracted. Following the principle of opening and closing, rejecting immature abstraction is as important as abstraction itself.
- A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- A module will be said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding)
- But the real challenge is anticipation, anticipating is hard In real world, when you anticipate the risk is high that:
    - We do anticipate variations that won't vary.: Always implement things when you actually need them, never when you just foresee that you need them. Developing and maintaining an abstraction has a cost and if we won't need it this cost is negative.

- The risk is also high that we don't anticipate the variation that will really be needed. But once the need for variation becomes real this is your developer responsibility to refactor and create the right abstractions and the right stable code that will act upon these abstractions

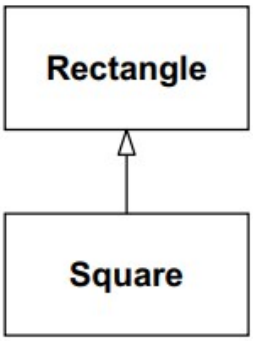
```
public class Circle { }

public class Square { }

public static class Drawer {
   public static void DrawShapes(IEnumerable<object> shapes) {
      foreach (object shape in shapes) {
         if (shape is Circle) {
            DrawCircle(shape as Circle);
         } else if (shape is Square) {
            DrawSquare(shape as Square);
         }
      }
   }
   private static void DrawCircle(Circle circle) { /*Draw
   circle*/ }

   private static void DrawSquare(Square square) { /*Draw
   Square*/ }
}

public interface IShape { void Draw(); }

public class Circle : IShape { public void Draw() { /*Draw
   circle*/ }}

public class Square : IShape { public void Draw() { /*Draw
   Square*/ } }

public static class Drawer {
   public static void DrawShapes(IEnumerable<IShape> shapes) {
      foreach (IShape shape in shapes) {
         shape.Draw();
      }
   }
}
```

LSP: The Liskov Substitution Principle:

- Subtypes must be able to completely replace their base types. The types satisfying the Liskov principle satisfy the following properties: if there is an object O2 of type T for every object O1 of type S, so that in all programs written for T, after replacing O2 with O1, the behavioral function of the program Change, then S is a subtype of T. The IS-A relationship in OOD is in terms of the behavior mode. The behavior mode can be reasonably assumed and is dependent on the client program. The effectiveness of the model can only be reflected through its client program. Some inheritance designs that violate the LSP can use the method of extracting common parts instead of inheritance.
- There are other, far more subtle, ways of violating the LSP. Consider an application which uses the Rectangle class

```
class Rectangle
{
   public: void SetWidth(double w) {itsWidth=w;}
   void SetHeight(double h) {itsHeight=w;}
   double GetHeight() const {return itsHeight;}
   double GetWidth() const {return itsWidth;}
private:
   Point itsTopLeft;
   double itsWidth;
   double itsHeight;
};
```

- Imagine that this application works well and is installed in many sites. As is the case with all successful software, its users demand changes from time to time. One day, the users demand the ability to manipulate squares in addition to rectangles.
- It is often said that inheritance is the IS-A relationship. In other words, if a new kind of object can be said to fulfill the IS-A relationship with an old kind of object, then the class of the new object should be derived from the class of the old object.
- For all normal intents and purposes, a square is a rectangle. Thus, it is logical to view the Square class as being derived from the Rectangle class.
- This use of the I -A relationship is sometimes thought to be one of the fundamental techniques of object-oriented analysis: A square is a rectangle, and so the Square class should be derived from the Rectangle class.



ISP: Interface Segregation Principle:

- Customers should not be forced to rely on the methods they do not use.
- Fat interfaces can cause abnormal and harmful coupling between their client programs. When a client program requires a change to the fat interface, it will affect other client programs. Therefore, client programs should only depend on what they actually call The method can achieve this goal by separating the fat interface into multiple specific client program interfaces.
- Each client-specific interface only declares the methods that its client program needs to call, and the implementation class implements all client- specific interfaces, releasing these coupling relationships so that the client programs do not depend on each other.
- Common methods of separating interfaces include the use of delegate separation and the use of multiple inheritance separation. The key to separating interfaces is to group the customers of the interface.
- Consider a security system. In this system, there are Door objects that can be locked and unlocked, and which know whether they are open or closed.

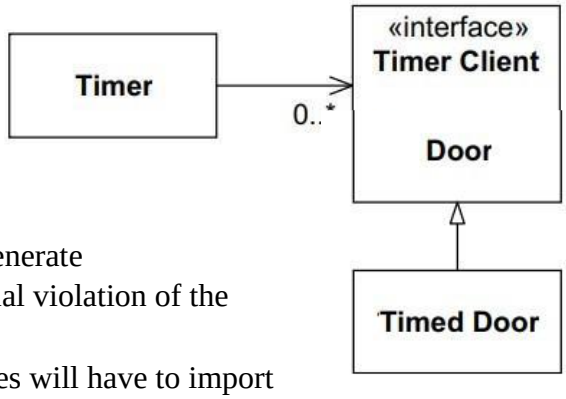
```
Security Door class Door
{
   public: virtual void Lock() = 0;
   virtual void Unlock() = 0;
   virtual bool IsDoorOpen() = 0;
};
```

- This class is abstract so that clients can use objects that conform to the Door interface, without having to depend on particular implementations of Door.
- Let's consider TimedDoor, needs to sound an alarm when the door has been left open for too long. In order to do this, the TimedDoor object communicates with another object called a Timer.

```
class Timer {
    public:
    void Register(int timeout, TimerClient* client);
};

class TimerClient {
    public:
    virtual void TimeOut() = 0;
};
```

- When an object wishes to be informed about a time-out, it calls the Register function of the Timer. The arguments of this function are the time of the time-out, and a pointer to a TimerClient object whose TimeOut function will be called when the time-out expires.

- We force Door, and therefore TimedDoor, to inherit from TimerClient. This ensures that TimerClient can register itself with the Timer and receive the TimeOut message.

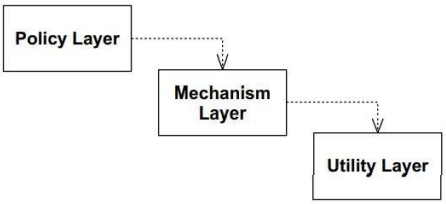


- Chief among these is that the Door class now depends on TimerClient. Not all varieties of Door need timing. Indeed, the original Door abstraction had nothing whatever to do with timing. If timing-free derivatives of Door are created, those derivatives will have to provide degenerate implementations for the TimeOut method-a potential violation of the LSP.
- Moreover, the applications that use those derivatives will have to import the definition of the TimerClient class, even though it is not used. That smells of Needless Complexity and Needless Redundancy

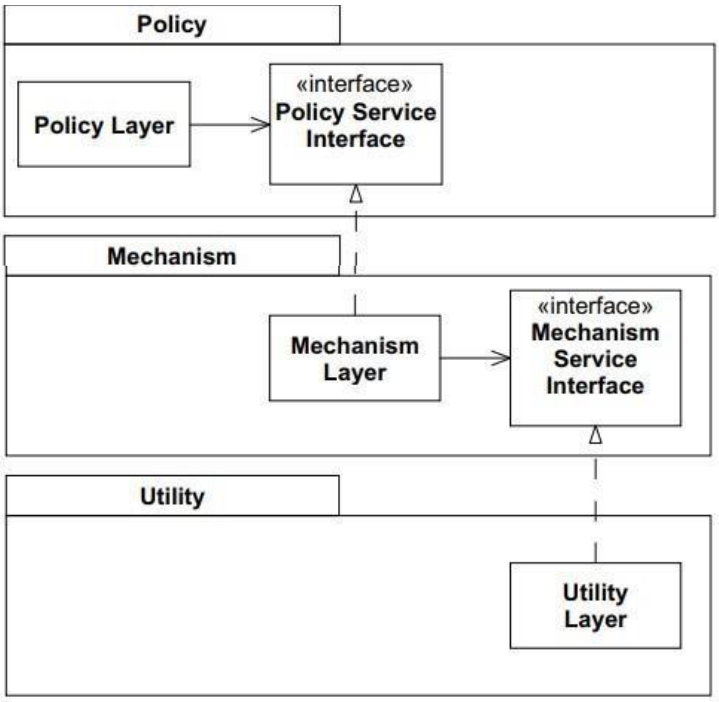
DIP: The Dependency Inversion Principle:

- High-level modules should not depend on low- level modules, both should rely on abstraction;
- Each higher level declares an abstract interface for the services it needs, the lower level implements these abstract interfaces, and each high-level class uses the next layer through this abstract interface, so that the high level does not depend on the lower level, but the lower level instead Depends on the abstract service interface declared by high-level.

Example:



- The high-level Policy layer uses a lower-level Mechanism layer, which in turn uses a detailed-level Utility layer. While this may look appropriate, it has the insidious characteristic that the Policy layer is sensitive to changes all the way down in the Utility layer.
- Dependency is transitive.
- The Policy layer depends on something that depends on the Utility layer; thus, the Policy layer transitively depends on the Utility layer. This is very unfortunate.
- Each of the upper-level layers declares an abstract interface for the services that it needs.
- The lower-level layers are then realized from these abstract interfaces
- This is sometimes known as the Hollywood principle: “Don’t call us, we’ll call you.” The lower-level modules provide the implementation for interfaces that are declared within, and called by, the upper-level modules.



- Abstraction should not depend on details, details should depend on abstractions;
- All the dependencies in the program should end in an abstract class or interface, that is:
    - No variable should hold a pointer or reference to a specific class;
    - No class should be derived from a concrete class;
    - No method should override the methods already implemented in any of its base classes;
- If a concrete class is unlikely to change, and other similar derived classes will not be created, then relying on it will not cause damage.
  For example: java language, String.
- Consider the case of the Button object and the Lamp object
- Figure shows a naive design. The Button object receives Poll messages, determines if the button has been pressed, and then simply sends the TurnOn or TurnOff message to the Lamp.