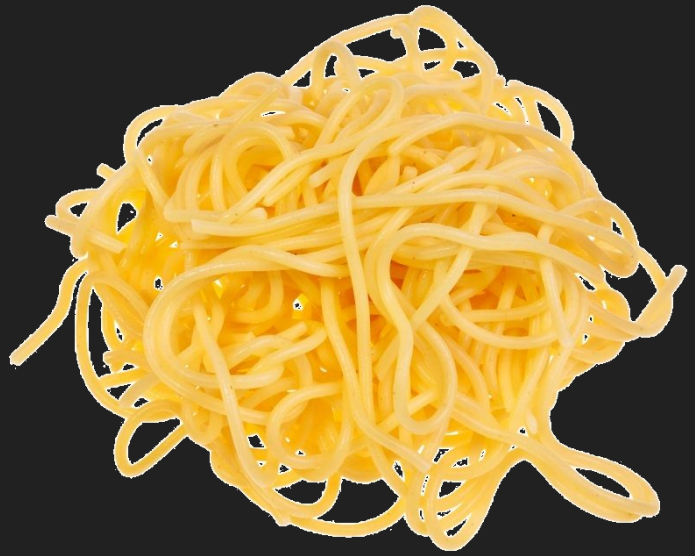
S.O.L.I.D Principles

Coding & Design Guidelines

Primary imperative

“Make things as simple as possible, but not simpler.”

Albert Einstein



S.O.L.I.D

S.O.L.I.D stands for:

- S - Single Responsibility Principle
- O - Open-closed Principle
- L - Liskov substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Single Responsibility Principle

*A class should have **one** and **only** one reason to change.*

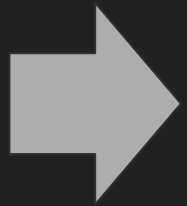
- Meaning that a class (but not only) should have only one job
- *Every module of a software system should do only one thing and do it well*
- Classes with names like “Manager”, “Controller”, “Util”, “Common” suggests too many responsibilities.

Single Responsibility Principle violation

```
class PhoneManager {
public:
   long getPhoneNumber() const;
   void makeACall(const long&);
   long receiveACall() const;
   void createAMessage(const std::string&);
   void sendAMessage(const std::string&);
   std::string readAMessage(const std::string&) const;
   void takePicture() const;
   void sendPicture(const Picture&);
   Picture& receivePicture() const;
   void browse();
   void initialize();
   void connectToNetwork();
}
```

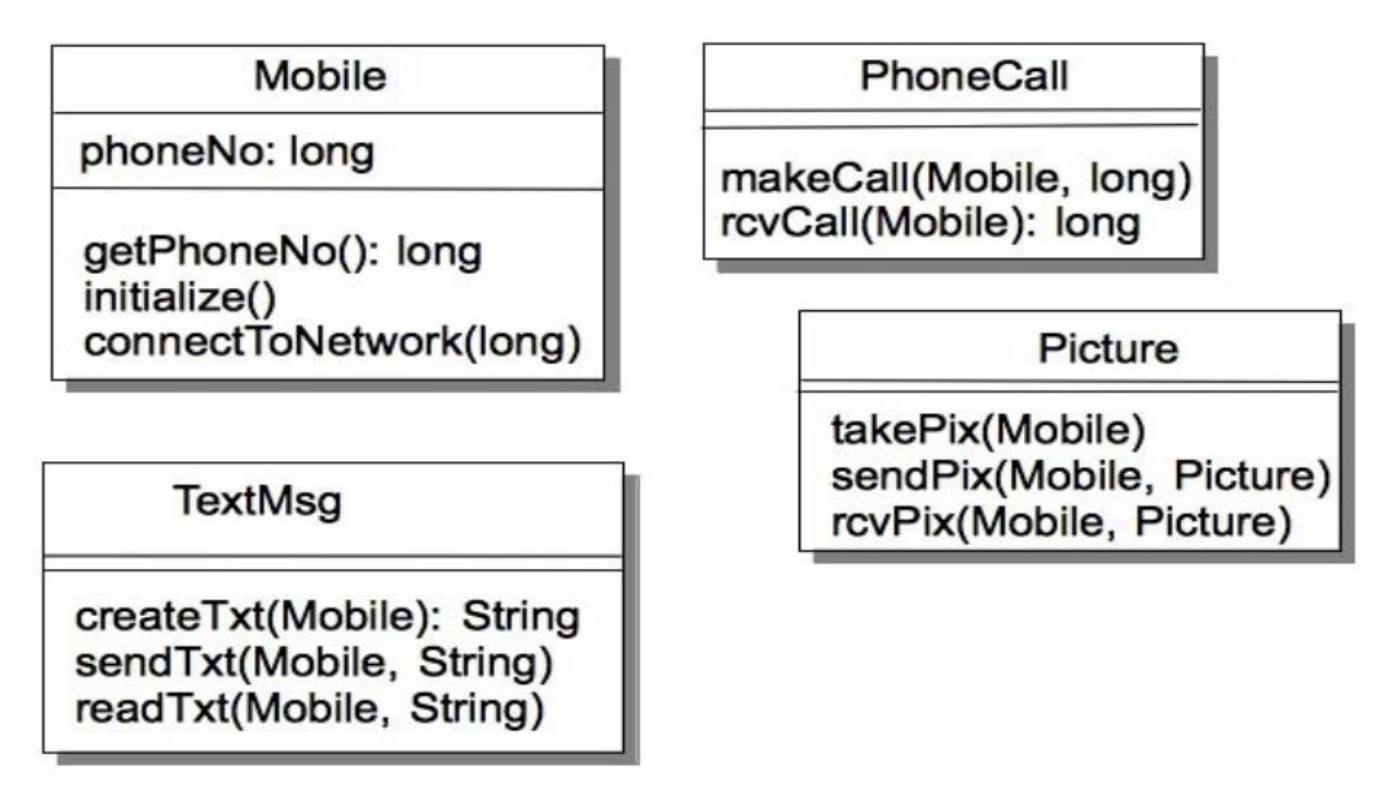
Single Responsibility Principle breaker contd.

```
class PhoneManager {
public:
   long getPhoneNumber() const;
   void makeACall(const long&);
   long receiveACall() const;
   void createAMessage(const std::string&);
   void sendAMessage(const std::string&);
   std::string readAMessage(const std::string&) const;
   void takePicture() const;
   void sendPicture(const Picture&);
   Picture& receivePicture() const;
   void browse();
   void initialize();
   void connectToNetwork();
}
```



```
class Phone {
public:
   void initialize();
private:
   Dialer dialer;
   Messenger messenger;
   PictureTaker pictureTaker;
   Browser browser;
   NetworkConnector networkConnector;
}
```

Single Responsibility Principle contd.



Open-closed principle

- Classes should be open for extension and closed for modification
- Find the behavior that does not vary and abstract that behavior up into a super/base class
- That locks the base code away from modification but all subclasses will inherit that behavior
- You are encapsulating the behavior that varies in the subclasses (those classes that extend the base class) and closing the base class from modification

Open-closed principle contd.

```
class ShapeContainer {
public:
   void addCircle(const Circle&);
   void addSquare(const Square&);
   void addRectangle(const Rectangle&);
   void getAllShapesAreasAndPerimeters()
const;
private:
   std::vector<Circle> circles;
   std::vector<Square> squares;
   std::vector<Rectangle> rectangles;
};
```

```
class ShapeContainer {
public:
   void addShape(Shape*);
   void getAllShapesAreasAndPerimeters()
const;
private:
   std::vector<Shape*> shapes;
};
```

Liskov Substitution Principle

- Subclasses must be substitutable for their base class
- Inheritance should be well designed and well behaved
- One of the best and canonical examples of violating the Liskov Substitution Principle is the Rectangle/Square example
- In any case a user should be able to instantiate an object as a subclass and use all the base class functionality invisibly

Liskov Substitution Principle contd.

```
class Rectangle {
private:
   double width;
   double height;
public:
   void setWidth(const double&);
   void setHeight(const double&);
   double getArea() const;
};

class Square : public Rectangle {
public:
   Square(const double& side) {
       setWidth(side);     // ???
       setHeight(side);    // ???
   }
};
```

Liskov Substitution Principle contd.

```
class Shape {
public:
   virtual double getArea() const = 0;
};

class Rectangle : public Shape {
private:
   double width;
   double height;
public:
   void setWidth(const double&);
   void setHeight(const double&);
   virtual double getArea() const override;

};
```

```
class Square : public Shape {
private:
   double side;
public:
   void setSide(const double&);
   virtual double getArea() const
override;
};
```

Interface Segregation Principle

- Clients shouldn’t have to depend on interfaces they don’t use. In particular, they shouldn’t have to *depend on methods they don’t use*
- Violation is done by start adding new methods to your interface because one of the subclasses that implements the interface needs it – and others do not
- How to fix this?

Interface Segregation Principle

- Clients shouldn’t have to depend on interfaces they don’t use. In particular, they shouldn’t have to *depend on methods they don’t use*
- Violation is done by start adding new methods to your interface because one of the subclasses that implements the interface needs it – and others do not
- How to fix this?
- The answer is: make more interfaces.
- Instead of adding new methods that are only appropriate to one or a few implementation classes, that you make a new interface

Interface Segregation Principle contd.

```
class Shape {
public:
    virtual double getArea() = 0;
    virtual double getVolume() = 0;
};

class Square : public Shape {
    virtual double getArea() override;    // square has
    virtual double getVolume() override; // square doesn't have
};
```

Interface Segregation Principle contd.

```
class Shape {
public:
    virtual double getArea() = 0;
    virtual double getVolume() = 0;
};

class Square : public Shape {
    virtual double getArea() override;    // square has
    virtual double getVolume() override; // square doesn't have
};
```

Interface Segregation Principle contd.

```
class TripleDimensionShape {
public:
   virtual double getVolume() = 0;
};

class DoubleDimensionShape {
public:
   virtual double getArea() = 0;
};
```

```
class Square : public DoubleDimensionShape {
   virtual double getArea() override;    // square has
};

class Cuboid : public DoubleDimensionShape, public
TripleDimensionShape {
   virtual double getVolume() override; // cuboid has
   virtual double getArea() override;    // cuboid has
};
```

Dependency inversion Principle

- Robert C. Martin introduced the Dependency Inversion Principle in his C++ Report
- *High-level modules should not depend on low-level modules. Both should depend on abstraction*
- *Abstractions should not depend on details. Details should depend on abstractions*
- The simple version of this is: don’t depend on concrete classes; depend on abstractions

Dependency inversion Principle contd.

```
class Processor {
public:
    void setOutputStream(const File&);
    void setOutputStream(const Console&);
private:
    InputStream inputStream;
    OutputStream outputStream;  // could be file, could be console
};
```

Dependency inversion Principle contd.

```
class OutputStream {
}

class FileOutputStream : public OutputStream {
}

class ConsoleOutputStream : public OutputStream {
}

class Processor {
public:
   void setOutputStream(OutputStream*) // combines both
private:
   InputStream inputStream;
   OutputStream* outputStream;
};
```

Other object oriented design principles

not solid, but still important

Encapsulate things in your design that are likely to change

- Separate features and methods that remain relatively constant through the program from those that will change
- Isolate the parts that will change a lot into a separate class (or classes) that we can depend on changing

Encapsulate things in your design that are likely to change contd.

```
class Violinist {
public:
   void setUpMusic();
   void tuneInstrument();
   void play();
};
```

Encapsulate things in your design that are likely to change contd.

```
class ViolinStyle {
public:
    virtual void getMusicStyle() = 0;
    virtual void play() = 0;
}

class Clasics : public ViolinStyle {
public:
    virtual void getMusicStyle() override;
    virtual void play() override;
}
```

```
class Violinist {
private:
    ViolinStyle* violinStyle;
public:
    void setUpMusic();
    void tuneInstrument();
    void
setMusicStyle(ViolinStyle*);
};
```

Don’t repeat yourself principle (DRY)

- Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- It’s been handed down ever since developers started thinking about better ways to write programs

Code to interfaces, rather than to implementation

- Interfaces are considered to be more stable than implementation
- When coding to interface, your program becomes easier to extend and modify

The Principle of Least Knowledge (PLK)

- Also known as *Law of Demeter*
- It says that classes should collaborate indirectly with as few other classes as possible

```
public:
    double getTemperature(const Sensor theSensor) {
        return theSensor.getSensorData().getOilData().getConditions().getTemperature();
    }
```

The Principle of Least Knowledge (PLK)

- Also known as *Law of Demeter*
- It says that classes should collaborate indirectly with as few other classes as possible

```
public:
    double getTemperature(const Sensor theSensor) {
        return theSensor.getSensorData().getOilData().getConditions().getTemperature();
    }
```

The Principle of Least Knowledge (PLK)

- Also known as *Law of Demeter*
- It says that classes should collaborate indirectly with as few other classes as possible

```
public:
    double getTemperature(const double theSensor) {
        return theSensor;
    }

.getTempearature(sensor.getTemperature());
```

References

- *Software Development and Professional Practice - John Dooley*
- *Clean Code - Robert C. Martin*