SRP (단일책임의 원칙: single responsibility principle)

```
#include <iostream>
#include <string>

class User
{
private:
    std::string name;//문자열 선언
    std::string email;

public:
    User(const std::string& name, const std::string& email) : name(name), email(email) {}

    std::string Name() const//클래스 함수,이름,이메일 반환
    {
        return name;
    }

    std::string Email() const
    {
        return email;
    }
};

class UserRepository
```

```
//단일 책임원칙 "하나의 클래스에 하나의 역활을 부여할것"
//기능을 분리하여 코드의 가독성과 유지보수를 용이하게 함.
{
public:
    void saveUser(const User& user)
    {
        // Save user to database
        std::cout << "User saved to database: " << user.Name() << std::endl;
    }
};

class EmailService
{
public:
    void sendWelcomeEmail(const User& user)
    {
        // Send welcome email to user
        std::cout << "Welcome email sent to: " << user.Email() << std::endl;
    }
};

class UserActivityLogger
{
public:
    void logUserActivity(const User& user)
```

```
    {
        // Log user activity
        std::cout << "Logging activity for user: " << user.Name() << std::endl;
    }
};

class UserService
{
private:
    UserRepository userRepository;
    EmailService emailService;
    UserActivityLogger userActivityLogger;

public:
    void registerUser(const User& user)
    {
        userRepository.saveUser(user);
        emailService.sendWelcomeEmail(user);
        userActivityLogger.logUserActivity(user);
    }
};

int main()
{
    User user("jjn", "naru4231@gmail.com");
```

```
    UserService userService;
    userService.registerUser(user);

    return 0;
}
```

하나의 클래스는 하나의 책임만 가져야 한다는 원칙. 클래스가 변경되는 이유는 단 하나여야 한다는 의미. 이 원칙을 지키면 클래스의 응집도가 높아지고, 유지 보수성이 향상됨. 함수의 이름을 보고 기능을 유추할수 있어 가독성도 좋아짐

OCP (개방폐쇄의 원칙: open close principle)

```
#include <iostream>

class Report//리포트 작성 부모 클래스
{
public:
    virtual void generate() = 0;//추상함수 "generate"
    virtual ~Report() = default;
};

class PDFReport : public Report
{
public:
    void generate() override
    //generate 추상함수를 각각 자녀클래스에서 구현
    //pdf생성 클래스에서는
    {
        std::cout << "Generating PDF report..." << std::endl;
```

```
    }
};

class HTMLReport : public Report
{
public:
    void generate() override
    {
        std::cout << "Generating HTML report..." << std::endl;
    }
};

class XMLReport : public Report
{
public:
    void generate() override
    {
        std::cout << "Generating XML report..." << std::endl;
    }
};

int main()
{
    Report* pdfReport = new PDFReport();
    pdfReport->generate();   // Generating PDF report...
```

```
    Report* htmlReport = new HTMLReport();
    htmlReport->generate();   // Generating HTML report...

    Report* xmlReport = new XMLReport();
    xmlReport->generate();   // Generating XML report...

    delete pdfReport;
    delete htmlReport;
    delete xmlReport;

    return 0;
}
```

소프트웨어 구성 요소는 확장에 대해서는 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다는 원칙. 즉, 새로운 기능을 추가할 수 있어야 하지만, 기존 코드를 수정하지 않고도 확장할 수 있어야 한다는 의미.

LSP (리스코브 치환의 원칙: the liskov substitution principle)

```
#include <iostream>

class Shape
{
public:
    virtual int getArea() = 0;//추상함수
    virtual ~Shape() = default;
};
```

```
class Rectangle : public Shape
{
private:
    int width;//속성은 높이와 너비
    int height;
public:
    Rectangle(int w, int h) : width(w), height(h) {}
    int getArea() override
    //OCP원칙을 사용하여 직사각형에서는 직사각형넓이 구하는 함수 구현
    {
        return width * height;
    }
};

class Square : public Shape
//정사각형이 직사각형에 속하기는 하나 정사각형클래스가 직사각형 클래스에 속하지는 x
//넓이를 구하는 getarea함수에서 자식클래스(정사각형)는 부모클래스(직사각형) 대체 불가
{
private:
    int side;//속성은 면 길이 하나
public:
    Square(int s) : side(s) {}
    int getArea() override
    //정사각형 클래스에서는 정사각형넓이 구하는 함수 구현
```

```
    {
        return side * side;
    }
};

class AreaCalculator
//구역 계산 클래스
//SRP원칙을 적용하여 구역을 계산하는 역활만 하는 클래스
{
public:
    void calculateArea(Shape* shape)
    {
        std::cout << "Area: " << shape->getArea() << std::endl;
    }
};

int main()
{
    AreaCalculator calculator;//구역을 계산하는 객체

    Shape* rectangle = new Rectangle(5, 4);//메모리 할당과 4x5 직사각형 생성
    calculator.calculateArea(rectangle);//20

    Shape* square = new Square(5);//메모리 할당과 5x5의 정사각형 생성
    calculator.calculateArea(square);//25
```

```
    delete rectangle;//메모리 해제
    delete square;

    return 0;
}
```

자식 클래스는 언제나 부모 클래스를 대체할 수 있어야 한다는 원칙. 즉, 프로그램에서 부모 클래스의 객체가 사용되는 모든 곳에서 자식 클래스의 객체로 바꿔도 문제가 발생하지 않아야 한다는 것. 이를 통해 상속 관계에서의 일관성을 유지할 수 있음.

ISP (인터페이스 분리의 원칙: interface segregation principle)

```
#include <iostream>

class Workable//일과 관련된 부모 클래스
{
public:
    virtual void work() = 0;
    virtual ~Workable() {}
};

class Eatable//먹는것과 관련된 부모 클래스
{
public:
    virtual void eat() = 0;
    virtual ~Eatable() {}
};
```

```
class Employee : public Workable, public Eatable
//다중 상속으로 employee의 역활 표현(workable,eatable)
{
public:
    void work() override //work함수 구현
    {
        std::cout << "Employee is working" << std::endl;
    }
    void eat() override //eat함수 구현
    {
        std::cout << "Employee is eating" << std::endl;
    }
};

class Robot : public Workable
{
public:
    void work() override //work 함수 구현
    {
        std::cout << "Robot is working" << std::endl;
    }
    //eatable에 상속되지 않았으므로 eat함수 구현필요 x
    //workable과 eatable클래스를 분리함으로서 로봇이 사용하지않는 인터페이스 구현x
};
```

```
int main()
{
	Workable* employee = new Employee();
	employee->work();
	dynamic_cast<Eatable*>(employee)->eat();

	Workable* robot = new Robot();
	robot->work();

	delete employee;
	delete robot;

	return 0;
}
```

특정 클라이언트를 위한 인터페이스는 그 클라이언트에 특화되어 있어야 한다는 원칙. 즉, 클라이언트가 사용하지 않는 메서드는 구현할 필요가 없다는 뜻. 이 원칙을 지키면 불필요한 의존성이 줄어듬.

DIP (의존성역전의 원칙: dependency inversion principle)

```
#include <iostream>

class Switchable
//켜고 끌수 있는 모든 함수에서 동작
//기존과 다르게 fan.spin,fan.stop을 사용하지 않기 때문에
//하위레벨의 모듈에 구속x
{
public:
```

```
    virtual void turnOn() = 0;//켜고 끄는 추상함수
    virtual void turnOff() = 0;
    virtual ~Switchable() {}
};

class Fan : public Switchable
{
public:
    void turnOn() override //fan클래스에서 함수 구현
    //fan 클래스이기 때문에 fan을 돌리고 멈추는걸로 구현
    {
        std::cout << "Fan is spinning" << std::endl;
    }
    void turnOff() override
    {
        std::cout << "Fan is stopping" << std::endl;
    }
};

class Switch//스위치 함수,메인문에서 작동
{
private:
    Switchable* device;//swithable속성을 가진 "device"
public:
    Switch(Switchable* device) : device(device) {}
```

```
    //device 메모리 할당

    void turnOn()
    //메인문에서 device를 켜고 끌때 사용하는 turn on
    {
        device->turnOn();//device 켜고 끄는 함수,추상함수 turn on
    }

    void turnOff()
    {
        device->turnOff();
    }
};

int main()
{
    Fan* fan = new Fan();//fan메모리 할당
    Switch switch1(fan);

    switch1.turnOn();
    switch1.turnOff();

    delete fan;//fan 메모리 해제
    return 0;
}
```

상위 모듈은 하위 모듈에 의존해서는 안 되며, 둘 다 추상화에 의존해야 한다는 원칙.
즉, 구체적인 구현이 아닌 추상화된 인터페이스에 의존함으로써 유연성과 재사용성이 향상.