SOLID DESIGN PRNCIPLES

Contents

1 Single Responsibility Principle

- A class should have only one reason to change.
- *Separation of Concerns* – different classes handling different, independent tasks or problems.

(Robert C. Martin)

Motivating Example

```
struct Journal //you create a journal class
{
  string title;
  vector<string> entries;
  // Constructor
  explicit Journal(const string& title)
    : title{title}
  {
  }
  // Adding entries to journal
  void add(const string& entry);

  // persistence is a separate concern
  void save(const string& filename);
};
```

We'll use the example of a Journal to demonstrate the SRP.

This journal has a title, and some entries and some methods – which would allow us to add entries to the journal and also, **save entries** to a file and consequently **load entries** from a file.

According to the SRP, saving the journal has to do with persistence, so persistence is a separate concern. Adding persistence to your journal would be violating the SRP.

Now imagine if you have, in addition to a journal, you also have a dozen other domain objects that you're working with and you give every one of them a function called SAVE and a function called LOAD etc etc. The problem with this is that when you want to change PERSISTENCE, you would have to change persistent in 10 to 20 classes which are actually using it.

```
// Implementation A - Non-SRP
void Journal::save(const string& filename)
{
  ofstream ofs(filename);
  for (auto& s : entries)
    ofs << s << endl;
}
```

For example, if one day you decide to stop using files and decide to use databases instead, you realise that you have to change this **everywhere**, which is not good. So essentially what we do is called a **separation of concerns.**

```
// Implementation B - SRP!
struct PersistenceManager
{
  static void save(const Journal& j, const string&
    filename)
  {
    ofstream ofs(filename);
    for (auto& s : j.entries)
      ofs << s << endl;
  }
};
```

Separation of Concerns – SRP Aligned Approach

So the Journal should take care of the journal entries and the journal title and its kind of storage mechanism with the strings and the counter, but if you want to PERSIST things, you NEED to have a separate component, a separate class. So we create a separate class to manage Persistence:

2 Open-Closed Principle

- Classes should be open for extension but closed for modification.

Motivating Example

Suppose you are making a website which sells certain products and your manager wants you to filter each of your products. Let's also suppose that each of the products has certain traits – like for example, colour and size.

```
10    enum class Color { red, green, blue };
11    enum class Size { small, medium, large };
12
13    struct Product
14    {
15      string name;
16      Color color;
17      Size size;
18    };
```

Your manager comes to you and tells you he would like you to filter by colour. So you make a new method for filtering by colour.

```
struct ProductFilter
{
  typedef vector<Product*> Items;
  // Colour filter
  Items by_color(Items items, const Color
    color)
  {
    Items result;
    for (auto& i : items)
      if (i->color == color)
        result.push_back(i);
    return result;
  }
```

After a preliminary round of analysis, the manager comes back to you and say "Can you *also* filter by size?" **And if you end up basically copying the code that you've written and pasting it and modifying it accordingly to adjust it by size like so – see image directly below, you have violated the Open-Closed Principle.**

```
Items by_size(Items items, const Size
  size)
{
  Items result;
  for (auto& i : items)
    if (i->size == size)
      result.push_back(i);
  return result;
}
```

```
Items by_size_and_color(Items items,
  const Size size, const Color color)
{
  Items result;
  for (auto& i : items)
    if (i->size == size && i->color ==
      color)
      result.push_back(i);
  return result;
}
```

Perhaps you've not only tested your previous editions of your code, but you've already shipped it to a client. This means you would introduce a **binary incompatibility** by adding another interface member to ProductFilter, which isn't great.

Somewhere down the line of course, your boss is going to come back to you once again and say: Can you make sure that our clients can filter by *both* size and colour? Then you go back to your existing code and start replicating code again.

Hmm, so you can begin to see how this is kinda problematic, because we have two criteria and e have three functions, if we had three criteria, we would have 8 functions. Generally, it doesn't scale and you don't really want to go into existing code.

Formal Definition
The OCP states that your systems **should be open to extensions,**

- ⇨ And this means that you should be able to extend systems **by inheritance**

and **closed to modification.**

- ⇨ What we demonstrated in the motivating example above is modifying existing code instead of inheriting and thereby extending the system.

An OCP Implementation

We create an abstract class that the BetterFilter can inherit from.

```
template <typename T> struct Filter
{
    // Returns a vector of tree pointers.
  virtual vector<T*> filter(vector<T*> items,
                            Specification<T>& spec) = 0;
};
```

```
struct BetterFilter : Filter<Product> // note that this is a Produ
{
  vector<Product*> filter(vector<Product*> items,
                          Specification<Product> &spec) override
  {
    vector<Product*> result;
    for (auto& p : items)
      if (spec.is_satisfied(p))
        result.push_back(p);
    return result;
  }
};
```

We start off by being really really generic – specify the vector of product pointers as the result, and then we check if the iterm in items conforms to the specification. If the specification is satisfied, we add it to the result.

Then specifiy the different metrics which we would like to search by: for instance – by colour: if we want to do a colour specification, we can define it on the left, and if we wanted a size specification we can define it on the right:

```
struct ColorSpecification :
  Specification<Product> // specialising
  template
{
  Color color;

  ColorSpecification(Color color) :
    color(color) {}

  bool is_satisfied(Product *item) const
    override {
    return item->color == color; //
  }
};
```

```
 // Same thing for Size
struct SizeSpecification :
  Specification<Product>
{
  Size size;

  explicit SizeSpecification(const Size size)
    : size{ size }
  {
  }


  bool is_satisfied(Product* item) const
    override {
    return item->size == size;
  }
};
```

To filter by both size and colour, we can create a combinator. To create this combinator, we will have to create the generic type first and adapt it to our product class later.

```
template <typename T> struct AndSpecification :
  Specification<T>
{
  const Specification<T>& first; //first metric
  const Specification<T>& second; // second
    metric
  // Constructor
  AndSpecification(const Specification<T>&
    first, const Specification<T>& second)
    : first(first), second(second) {}

  bool is_satisfied(T *item) const override {
      // Note you already overrided these
        is_satisfieds below.
    return first.is_satisfied(item) &&
      second.is_satisfied(item);
  }
};
```

In Sum

The goal of the open close principle is to avoid having to jump into code that you've already written. That's it.

Closing something from modification suggests that we would never have to go back into the filter interface or the Specification interface, you wouldn't have to change those, and you would extend them by inheritance and then you would create maybe these combinators like the AndSpecification on the left.

Note that we also used templated arguments, so we are also not constrained ourselves to the Product Class. We can also have other kinds of items here, and its all very nice and flexible and very usable as well.

---

3 Liskov Substitution Principle

- You should be able to substitute a base type for a subtype without altering the correctness of the program.

Motivating Example

We'll use the following rectangle class for this example.

```
class Rectangle
{
protected: // cannot be accessed outside class,
  can be accessed by child class
  int width, height;
public:
  Rectangle(const int width, const int height) //
    Constructor
    : width{width}, height{height} { }

    // Create getters and setters
  int get_width() const { return width; }
  virtual void set_width(const int width) { this-
    >width = width; }
  int get_height() const { return height; }
  virtual void set_height(const int height) {
    this->height = height; }

  int area() const { return width * height; }
};
```

Somewhere down the line, I decide that I want to extend this model, but before I do, let's suppose I'm already using some sort of API for working with Rectangle. So let's suppose there is some function called *process* which takes a rectangle reference to a rectangle.

```
void process(Rectangle& r)
{
  int w = r.get_width();
  r.set_height(10); // sets height to 10

  std::cout << "expected area = " << (w * 10)
    << ", got " << r.area() << std::endl; //
      expects area equal to width * 10.

}
```

Somewhere along the workflow we decide to use **inheritance** to inherit from Rectangle to get a Square.

```
class Square : public Rectangle
{
public:
  Square(int size): Rectangle(size,size) {} //
    constructor
  void set_width(const int width) override {
    this->width = height = width;
  }
  void set_height(const int height) override {
    this->height = width = height;
  }
};
```

And by doing this we actually violated the LSP. It states that if you have a function which takes a base class, any **derived class should be able to be substituted into this function without any problems.**

Geometrically, this makes sense – a square is a **special case of a rectangle.** But it is NOT a good idea to inherit a square from a rectangle. However, to adhere to the LSP, you can do it differently.

LSP Approach

Create a Boolean variable within the class Rectangle for instance and set it to true if it's a square, and false if not. However, you can do this instead – and this too doesn't break the LSP:

```
struct RectangleFactory
{
  static Rectangle create_rectangle(int w, int
    h); // returns a rectangle
  static Rectangle create_square(int size); //
    returns a square.
}; // doesn't break the LSP!!
```

Notice that none of the methods of the Base Class *Rectangle* are overrided. Instead, the RectangleFactory class helps to bring both classes together. This doesn't break the LSP.

4 Interface Segregation Principle

- Don't put too much into an interface – split into separate interfaces
- *YAGNI* – You Ain't Going to Need It.

Motivating Example
The idea of the LSP is basic ally to get you to **NOT** create interfaces which are too large and which require the implementors to implement *too much.*

Consider you have a document of some type and you have a multifunction printer called the *IMachine:*

This IMachine is really cool, it's a multifunction printer, which can in fact not just print, but scan and fax documents, there's no problem in using this interface.

```
struct Document;

struct IMachine
{
  virtual void print(Document& doc) = 0;
  virtual void fax(Document& doc) = 0;
  virtual void scan(Document& doc) = 0;
};
```

```
struct Scanner: IMachine
{
    void print(Document& doc) override; // throw
      some exception? Leave them as null?
    void fax(Document& doc) override; // throw
      some exception? Leave them as null?
    void scan(Document& doc) override; // the
      only useful method!
};
```

The problem is though is that what happens if you want to implement **only** a printer / scanner / fax machine?

If we want the scanner for example, and we only have the IMachine to inherit from, then we'll do the following – we'll overwrite all three methods: buit what do we even write for the print() and fax() methods. Do you leave them as **null?** Throw some exception?

Either way you are telling a client that you are giving them a Scanner which can also print, which isn't true. But you're giving them the API anyway, simply because the interface that's been defined for handling all this stuff is just too big.

So what we really need to do is to segregate the interfaces because no client would require always all printing scanning and faxing.

Instead of the above approach, we would use the following implementation (right) which is more aligned to the ISP:

Note that we create **3 separate interfaces,** one for scanning one for printing and one for faxing.

From these abstractions, we can create a printer:

```
struct Printer : IPrinter
{
  void print(Document& doc) override;
};
```

```
struct IPrinter //takes care of only printing
  the document
{
  virtual void print(Document& doc) = 0;
};

struct IScanner // takes care of only scanning
  the document
{
  virtual void scan(Document& doc) = 0;
};
struct IFax // takes care of only faxing the
  document.
{
    virtual void fax(Document& doc) = 0;
};
```

Or even a scanner without too many additional classes or methods:

```
struct Scanner : IScanner
{
  void scan(Document& doc) override;
};
```

If you would like to make a more complex machine, IMachine should inherit from one or more of these interfaces. For instance,

```
struct IMachine: IPrinter, IScanner
{
}; // keep these abstract (no body required. )
```

The goal of the ISP is to make our interfaces smaller so that whenever a client uses one of these chunks of the interface, **it's all coherent!** They don't have to throw exceptions out of methods or anything of that sort.

---

5 Dependency **Inversion** Principle (Different from Dependency Injection!)

- High level modules should not depend upon low-level ones, use abstractions instead.

Motivating Example

```
enum class Relationship
{
  parent,
  child,
  sibling
};

struct Person
{
  string name;
};
```

We can model relationships between people and we define a Person by only using their name:

We want to get a low level construct for actually storing the relationships. This is an abstraction. So we do this by creating a struct for Relationships that inherits from our relationship browser.

```
struct Relationships : RelationshipBrowser // low-
  level module
// note that relationships inherits the
  relationshipbrowser interface.
// low level construct in our design.
{
  vector<tuple<Person, Relationship, Person>>
    relations;

  void add_parent_and_child(const Person& parent,
    const Person& child)
  {
    relations.push_back({parent,
      Relationship::parent, child});
    relations.push_back({child,
      Relationship::child, parent});
  }
```

Now let's try to find how we can do *Research* on relationships, and this is a high level module.

Wrong Approach

So this is something that you **should not do.** Remember that high level modules should not depend on low level modules – we'll see what happens when there is some form of dependency:

```
Research(const Relationships& relationships)
{
  auto& relations = relationships.relations; /
    / the high level module is now dependent
    on the low level module.
  // If I want to find all the relations of
    John, we can certainly do it:
  for (auto&& [first, rel, second] : relations)
  {
    if (first.name == "John" && rel ==
      Relationship::parent)
    {
      cout << "John has a child called " <<
        second.name << endl;
    }
  }
}
```

Note that this produces the results that we want. However, let's imagine what happens if, for example, the low level module decides to change the storage of relations, like it could…

- Make the **relations variable** private? And it tries to hide it completely?

In this case, the high level code is completely broken, unfortunately. So how do you fix this in this situation?

DIP Aligned Approach

Typically we use *abstractions* – so instead of having the direct dependency on some low level module you would introduce an abstraction – like some sort of interface, but that's only a part of the problem.

The other part of the problem is really if you want to find all the children of a particular person, I would argue that the **single responsibility principle** still puts this responsibility as part of the relationships class.

So your research doesn't really have to delve into, you know, finding the children of somebody. You can move the finding children method into a lower level module, or by introducing an abstraction.

```
struct RelationshipBrowser
{
  virtual vector<Person> find_all_children_of(cons
    string& name) = 0;
};
```

So this abstraction is going to be called a **RelationshipBrowser.** Our RelationshipBrowser is going to provide a few useful bits of functionality for searching for all the children of a particular person.

So now, instead of depending on low level modules, we will depend on just an abstraction – the Relationship Browser.

```
Research(RelationshipBrowser& browser) //
  constructor
{
  for (auto& child :
    browser.find_all_children_of("John"))
  {
    cout << "John has a child called " <<
      child.name << endl;
  }
}
```

We hope to have dependencies on interfaces rather than concrete implementations. And you don't want to delve into other classes details and start depending on their details, because remember, at some point in time, relationships might decide to replace the vector by some kind of map or something. And in this case, you're going to be really screwed because you're iterating over a vector of tuples and there's no longer a vector of tuples for you to iterate.

Dependency Injection allows you to specify certain defaults. For instance, it will allow you to specify that whenever someone wants a RelationshipBrowser, they SHOULD be provided a copy or a reference to the Relationships class. This is how dependency injection plays ON TOP of the dependency inversion principle, but the inversion principle just basically says that both of the high level and low level modules should be depending on ABSTRACTIONS and you should not depend on details of somebody else's implementation.