Coding Design Principles including UML, SOLID Principles, and Design Patterns

Syed Paymaan Raza

November 26, 2017

Contents

I've been reading a lot recently about what is considered good versus bad coding design. This sometimes is also referred to as *patterns* and *anti-patterns* respectively. What I've found is that it comes down to "our experience of things that made life easier". On the contrary, it's also "things that made life terrible for us – making trivial tasks harder than they actually should be". In some cases, there may be some overlap i.e. what person A considers as a good pattern is considered an anti-pattern by person B. In such cases, it usually boils down to what is the task at hand, what are the requirements, do I need a complicated data structure?, do I need a fancy pattern? etc. As always, in engineering, it's a matter of taste as well.

Note that most of the principles are language agnostic.

Before getting started, here's a list of resources I found useful when compiling this document:

1. C++ Core Guidelines
2. From STUPID to SOLID Code!
3. The SOLID Principles
4. SOLID – Object Oriented Design
5. SOLID Principles with real world examples
6. SOLID Principles with C# Examples

Note that I'll keep adding to this list as I discover more stuff.

One other thing that I learnt was that most problems in engineering can be solved by abstraction and indirection. Abstraction also involves simplifying a complicated problem into smaller easily solved problems.

Enough talk, let's get to the coding now!

1 SOLID

In general, we are trying to avoid some common themes that are considered as "anti-patterns" in coding:

1. Tight Coupling – components of your program have too many dependencies on each other. Changing one component affects other components. Such tightly coupled components are difficult to reuse because they most probably can't live on their own due to dependencies. That also makes them harder to test.

2. Duplication – DRY (Don't repeat yourself): On a basic level, use functions to abstract out common stuff. Note that, these functions should be granular to the point where one function does one thing only, just like a class should be designed to do. Functions should, ideally, also not have side-effects and be pure i.e. invoking the function with the same inputs multiple times should result in the same thing. Such pure functions don't depend on any global state.

3. Indescriptive naming – Naming stuff is art! Too short and it's not to descriptive; too long and it's hard to read; In general, keep the length of variables, functions, classes as minimum so long as that they describe what they intend to do. Also, functions should be verbs, classes should be nouns or verbs since they have both data and state. I generally name my classes as nouns since that's more natural to me.

I'm sure there are other anti-patterns worth discussing but that's a list to start with. I'll keep updating it as I find more things worth adding here.

Now let's get to SOLID, a set of design principles that help alleviate some of the issues listed above.

SOLID is an acronym; let's go letter by letter:

1.1 S - Single Responsibility Principle (SRP)

"Every class should have a single responsibility. There should never be more than one reason for a class to change."

Let's look at an example:

```
#include <iostream>
#include <vector>

class Shape {
  public:
    virtual ~Shape() = 0;
};
```

```
Shape::~Shape() {}

class Circle : public Shape {
  public:
    Circle(int r)
        : radius(r) {}
    int get_radius() const {
        return radius;
    }

  private:
    int radius;
};

class Square : public Shape {
  public:
    Square(int l)
        : length(l) {}
    int get_length() const {
        return length;
    }

  private:
    int length;
};

class AreaCalculator {
  public:
    AreaCalculator(const std::vector<Shape*>& v)
        : shapes(v) {}
    int sum() {
        return 1234; // some logic to calculate sum based on shapes
    }
    void output() {
        std::cout << "The sum is: " << sum() << std::endl;
    }

  private:
    std::vector<Shape*> shapes;
};
```

```
int main() {
    Circle c(5);
    Square s(10);
    AreaCalculator calc({&c, &s});
    calc.output();
    return 0;
}
```

```
The sum is: 1234
```

Here, we can see that `AreaCalculator` can calculate the sum of its shapes AND also output the sum.

Now what if instead of outputting to stdout, we want to serialize this to a JSON file?

Here, SRP is broken because `AreaCalculator` will have to be changed either if we decide to change the sum logic, or if we decide to change the output format. Instead, we can do something like this:

```
#include <iostream>
#include <vector>

class Shape {
  public:
    virtual ~Shape() = 0;
};

Shape::~Shape() {}

class Circle : public Shape {
  public:
    Circle(int r)
        : radius(r) {}
    int get_radius() const {
        return radius;
    }

  private:
    int radius;
};
```

```
class Square : public Shape {
  public:
    Square(int l)
        : length(l) {}
    int get_length() const {
        return length;
    }

  private:
    int length;
};

class AreaCalculator {
  public:
    AreaCalculator(const std::vector<Shape*>& v)
        : shapes(v) {}
    int sum() const {
        return 1234; // some logic to calculate sum based on shapes
    }

  private:
    std::vector<Shape*> shapes;
};

class AreaOutputter {
  public:
    AreaOutputter(const AreaCalculator& a)
        : calc(a) {}
    void output1() {
        std::cout << "The sum is: " << calc.sum() << std::endl;
    }
    void output2() {
        std::cout << "Another way of outputting sum is: " << calc.sum()
                  << std::endl;
    }

  private:
    const AreaCalculator& calc;
};
```

```
int main() {
    Circle c(5);
    Square s(10);
    AreaCalculator calc({&c, &s});
    AreaOutputter outputter(calc);
    outputter.output2();
    return 0;
}
```

```
Another way of outputting sum is: 1234
```

Note that although we discussed classes here, SRP can be applied to functions as well i.e. functions should only have one reason to change.

1.2 O - Open/Closed Principle (OCP)

"Objects or entities should be open for extension, but closed for modification."

1.2.1 Example 1

Copied from `scotch.io`:

This simply means that a class should be easily extendable without modifying the class itself. Let's take a look at the AreaCalculator class, especially it's sum method.

```
public function sum() {
    foreach($this->shapes as $shape) {
        if(is_a($shape, 'Square')) {
            $area[] = pow($shape->length, 2);
        } else if(is_a($shape, 'Circle')) {
            $area[] = pi() * pow($shape->radius, 2);
        }
    }

    return array_sum($area);
}
```

If we wanted the sum method to be able to sum the areas of more shapes, we would have to add more if/else blocks and that goes against the Open-closed principle since we have to modify `sum()` whenever we have a new shape i.e. it's not close to modification. Instead, we can make it extensible.

A way we can make this sum method better is to remove the logic to calculate the area of each shape out of the sum method and attach it to the shape's class.

```
class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }

    public function area() {
        return pow($this->length, 2);
    }
}
```

The same thing should be done for the `Circle` class, an area method should be added. Now, to calculate the sum of any shape provided should be as simple as:

```
public function sum() {
    foreach($this->shapes as $shape) {
        $area[] = $shape->area();
    }

    return array_sum($area);
}
```

Now we can create another shape class and pass it in when calculating the sum without breaking our code. However, now another problem arises, how do we know that the object passed into the AreaCalculator is actually a shape or if the shape has a method named area?

Coding to an interface is an integral part of S.O.L.I.D, a quick example is we create an interface, that every shape implements:

```
interface ShapeInterface {
    public function area();
```

```
}

class Circle implements ShapeInterface {
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }

    public function area() {
        return pi() * pow($this->radius, 2);
    }
}
```

In our `AreaCalculator` sum method we can check if the shapes provided are actually instances of the `ShapeInterface`, otherwise we throw an exception:

```
public function sum() {
    foreach($this->shapes as $shape) {
        if(is_a($shape, 'ShapeInterface')) {
            $area[] = $shape->area();
            continue;
        }

        throw new AreaCalculatorInvalidShapeException;
    }

    return array_sum($area);
}
```

1.2.2 Example 2

Use Strategy Design Pattern

1.2.3 Example 3

Use Visitor Design Pattern

1.3 L - Liskov Substitution Principle (LSP)

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program."

1.3.1 Example 1

In languages with compile-time type checking, LSP is pretty obvious since we'll catch any violations at compile-time:

```
class Vehicle {

    function startEngine() {
        // Default engine start functionality
    }

    function accelerate() {
        // Default acceleration functionality
    }
}

class Car extends Vehicle {

    function startEngine() {
        $this->engageIgnition();
        parent::startEngine();
    }

    private function engageIgnition() {
        // Ignition procedure
    }

}

class ElectricBus extends Vehicle {

    function accelerate() {
        $this->increaseVoltage();
        $this->connectIndividualEngines();
    }
```

```
    private function increaseVoltage() {
        // Electric logic
    }

    private function connectIndividualEngines() {
        // Connection logic
    }

}

class Driver {
    function go(Vehicle $v) {
        $v->startEngine();
        $v->accelerate();
    }
}
```

1.3.2 Example 2

In dynamically typed languages, we can get run-time errors. Copied from `scotch.io`:

Still making use of out `AreaCalculator` class, say we have a `VolumeCalculator` class that extends the `AreaCalculator` class:

```
class VolumeCalculator extends AreaCalulator {
    public function __construct($shapes = array()) {
        parent::__construct($shapes);
    }

    public function sum() {
        // logic to calculate the volumes and then return and array of output
        return array($summedData);
    }
}

class SumCalculatorOutputter {
    protected $calculator;

    public function __constructor(AreaCalculator $calculator) {
        $this->calculator = $calculator;
```

```
    }

    public function JSON() {
        $data = array(
            'sum' => $this->calculator->sum();
        );

        return json_encode($data);
    }

    public function HTML() {
        return implode('', array(
            '',
                'Sum of the areas of provided shapes: ',
                $this->calculator->sum(),
            ''
        ));
    }
}
```

If we tried to run an example like this:

```
$areas = new AreaCalculator($shapes);
$volumes = new AreaCalculator($solidShapes);

$output = new SumCalculatorOutputter($areas);
$output2 = new SumCalculatorOutputter($volumes);

$output2.HTML() // error!
```

The program does not squawk, but when we call the `HTML` method on the $output2 object we get an E_NOTICE error informing us of an array to string conversion.

To fix this, instead of returning an array from the VolumeCalculator class sum method, you should simply:

```
public function sum() {
    // logic to calculate the volumes and then return and array of output
    return $summedData; // summed data as float, double, or integer
}
```

1.4 I - Interface Segregation Principle (ISP)

"A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use."

1.4.1 Example 1

Copied from `scotch.io`:

Still using our shapes example, we know that we also have solid shapes, so since we would also want to calculate the volume of the shape, we can add another contract to the `ShapeInterface`:

```
interface ShapeInterface {
    public function area();
    public function volume();
}
```

Any shape we create must implement the `volume` method, but we know that squares are flat shapes and that they do not have volumes, so this interface would force the Square class to implement a method that it has no use of.

ISP says no to this, instead you could create another interface called `SolidShapeInterface` that has the volume contract and solid shapes like cubes e.t.c can implement this interface:

```
interface ShapeInterface {
    public function area();
}

interface SolidShapeInterface {
    public function volume();
}

class Cuboid implements ShapeInterface, SolidShapeInterface {
    public function area() {
        // calculate the surface area of the cuboid
    }

    public function volume() {
        // calculate the volume of the cuboid
    }
}
```

This is a much better approach, but a pitfall to watch out for is when type-hinting these interfaces, instead of using a `ShapeInterface` or a `SolidShapeInterface`.

You can create another interface, maybe `ManageShapeInterface`, and implement it on both the flat and solid shapes, this way you can easily see that it has a single API for managing the shapes. For example:

```
interface ManageShapeInterface {
    public function calculate();
}

class Square implements ShapeInterface, ManageShapeInterface {
    public function area() { /*Do stuff here*/ }

    public function calculate() {
        return $this->area();
    }
}

class Cuboid implements ShapeInterface, SolidShapeInterface, ManageShapeInterface {
    public function area() { /*Do stuff here*/ }
    public function volume() { /*Do stuff here*/ }

    public function calculate() {
        return $this->area() + $this->volume();
    }
}
```

Now in `AreaCalculator` class, we can easily replace the call to the area method with calculate and also check if the object is an instance of the `ManageShapeInterface` and not the `ShapeInterface`.

1.5 D - Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions."

1.5.1 Example 1

Copied from `scotch.io`: This might sound bloated, but it is really easy to understand. This principle allows for decoupling, an example that seems like the best way to explain this principle:

```
class PasswordReminder {
    private $dbConnection;

    public function __construct(MySQLConnection $dbConnection) {
        $this->dbConnection = $dbConnection;
    }
}
```

First the MySQLConnection is the low level module while the PasswordReminder is high level, but according to the definition of D in S.O.L.I.D. which states that Depend on Abstraction not on concretions, this snippet above violates this principle as the PasswordReminder class is being forced to depend on the MySQLConnection class.

Later if you were to change the database engine, you would also have to edit the PasswordReminder class and thus violates Open-close principle.

The PasswordReminder class should not care what database your application uses, to fix this again we "code to an interface", since high level and low level modules should depend on abstraction, we can create an interface:

```
interface DBConnectionInterface {
    public function connect();
}
```

The interface has a connect method and the MySQLConnection class implements this interface, also instead of directly type-hinting MySQLConnection class in the constructor of the PasswordReminder, we instead type-hint the interface and no matter the type of database your application uses, the PasswordReminder class can easily connect to the database without any problems and OCP is not violated.

```
class MySQLConnection implements DBConnectionInterface {
    public function connect() {
        return "Database connection";
    }
}
```

```
class PasswordReminder {
    private $dbConnection;

    public function __construct(DBConnectionInterface $dbConnection) {
        $this->dbConnection = $dbConnection;
    }
}
```

According to the little snippet above, you can now see that both the high level and low level modules depend on abstraction.

1.5.2 Example 2

A fantastic example of DIP which uses EReader and PDFBook can be found here.

2 UML

I think for UML (Universal Modeling Language), there are many resources on the internet that give good pictorial description of things. This section will basically be just a collection of such resources along with some of my personal notes based on what I've read on blogs, stack overflow, seen at work etc. Notice, however, that I'll just focus on the OOP and class diagram portion of UML, probably also a bit sequence diagrams. I *think* UML is much more than that as can be found here.

Some good links:

1. UML Cheat Sheet
2. UWashington Notes - Class Diagrams
3. UWashington Notes - Sequence Diagrams
4. NTUT Notes
5. UML with C++ examples
6. Codeproject UML
7. UML with more C++ examples

The above links also explains the difference between association, aggregation, and composition.

2.1 Notation and implementation in C++

White diamond: aggregation (weak composition) - $\text{shared}_{\text{ptr}}$, reference + you may or may not allocate the memory

Just arrow with black filled arrow head: has a reference to.. can be $\text{shared}_{\text{ptr}}$ or just a reference but you did not allocate the memory.

Black filled diamond: composition (strong) - $\text{unique}_{\text{ptr}}$ (own memory), reference

Just arrow with white filled (or unfilled) arrow head: inheritance.. sometimes we use dotted line if inheritance is implementing an interface.

Simple arrow with no diamond or white filled triangle generally can mean different but related things and it's annotated e.g. "creates" is written besides the arrow. Generally it means some sort of dependency.

- use pointers when we need dynamic switching
- use pointers when we need to encapsulate an abstract class (polymorphic runtime type).. although we can have references to abstract types as well which point to a concerete runtime object.
- use $\text{shared}_{\text{ptr}}$ for weak composition because pointee can exist without pointer
- use $\text{unique}_{\text{ptr}}$ for strong composition because pointee can not live without pointer
- $\text{unique}_{\text{ptr}}$ also enforces ownership so strong composition makes sense with it.

3 Design Patterns

3.1 Cheat sheets

- dzone design patterns ref card (web site)
- dzone design patterns ref card (document)
- gang of four ref card (document)
- design patterns ref card (document)

3.2 Terms and concepts

- OOP is an abstraction; can hide information and expose using desired APIs.
- OOP APIs are generally based on contracts.
- Interfaces are fundamental in OOP; objects are known only through their interfaces.
- An interface is only meant for communication to the outside world; internally, we can have different implementations (concreteations) for the same interface.
- Interfaces are abstract classes and therefore can' be instantiated. Their subclass implementations can though.
- Implementations should depend on interfaces, not the other way around.
- Design patterns often specify relationship between interfaces.
- Objects are created by instantiating a class; the object is an instance of the class. We also use the terms Instantiator (one who instantiates) and Instantiatee (object getting instantiated). This is generally depicted using a dashed arrowhead from Instantiator to Instantiatee.
- Object class and its type are different things. Class defines how the object is implemented. Type only refers to the common interface which the object can use e.g. obj.foo(). An object can have many types, and objects of different classes can have the same type.
- It's recommended to program to an interface, not an implementation. Two interfaces should talk to each other as well. This decouples implementation logic in different parts of the system.
- There are two benefits to manipulating objects solely based on interface defined by abstract classes:
  - Clients remain unaware of the specific types of object they use, as long as objects adhere to the interface that clients expect.
  - Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

- Reusing in OOP can be done using inheritance and composition. Inheritance is referred to as "white box" reuse since the child class knows about its parent (public info). Composition is "black box" reuse since the class that composes knows nothing about the composed object itself. Having said that, the class that composes generally allocates the composed obect so in we can say it "owns" the object.
- In general, prefer composition to inheritance:
  - Less coupling between two classes
  - Inheritance can break encapsulation
  - In inheritance, we can't change implementation at run-time like we can with compisition
- Delegation; objects can delegate their task to another object.
- Another third way of reuse is parameterized types / generics / templates in C++.
- Design for change in the system; changing one parts of the system shouldn't propagate to other parts so we don't have to change much; this is because we abstract and hide out functionality well. Here are some common causes of redesign:
  - Creating an object by specifying a class explicitly. It commits you to a particular implementation instead of a particular interface. (Abstract factory, Factory method, Prototype)
  - Dependence on specific operations. (Chain of Responsibility, Command)
  - Dependence on hardware and software platform. (Abstract factory, Bridge)
  - Dependence on object representations or implementations. Clients that know how an object is represented, stored, or located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading. (Abstract factory, Bridge, Memento, Proxy)
  - Algorthmic dependencies
  - Tight coupling. Classes that are tightly coupled are hard to reuse in isolation since they depend on each other.

  - Extending functionality by subclassing. This has implementation overhead and reduced isolation/abstraction.

- 3 Kinds of design patterns based on purpose:
  - Creational (C): Creational patterns concern the process of object creation.
  - Behavioaral (B): Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.
  - Structural (S): Structural patterns deal with the composition of classes or objects.
- OOP classes mostly nouns, functions verbs.
- OOP classes should be responsible for one thing only, similar to functions should do one thing only.
- Separation of concerns.
- Design should be closed for modification but open for extension.

3.3 Singleton (C)

Ensure a class only has one instance and provide a global point of access to it.

```
#include <iostream>
#include <memory>

using namespace std;

class MySingleton {
  public:
    static shared_ptr<MySingleton> get_instance() {
        if (!instance)
            instance =
                shared_ptr<MySingleton>(new MySingleton());
        return instance;
    }

    void foo() {
        cout << "calling foo..\n";
```

```
    }

  private:
    MySingleton(){};
    static shared_ptr<MySingleton> instance;
};

shared_ptr<MySingleton> MySingleton::instance = nullptr;

int main() {
    auto my_singleton_obj = MySingleton::get_instance();
    my_singleton_obj->foo();
    return 0;
}

calling foo..
```

Great. But what if we are in a multi-threaded environment? How do we ensure that all threads get only a single instance of the class? Here's one way using C++11:

```
#include <iostream>
#include <memory>
#include <mutex>

#include <mutex>

using namespace std;

class MySingleton {
  public:
    static void get_instance_helper() {
        if (!instance) {
            instance =
                shared_ptr<MySingleton>(new MySingleton());
        }
    }

    static shared_ptr<MySingleton> get_instance() {
        call_once(singleton_flag, get_instance_helper);
```

```
        return instance;
    }

    void foo() {
        cout << "calling foo..\n";
    }

  private:
    MySingleton(){};
    static once_flag singleton_flag;
    static shared_ptr<MySingleton> instance;
};

once_flag MySingleton::singleton_flag;
shared_ptr<MySingleton> MySingleton::instance = nullptr;

int main() {
    auto my_singleton_obj = MySingleton::get_instance();
    my_singleton_obj->foo();
    return 0;
}

calling foo..
```

Previously calls to `get_instance` weren't synchronized, but now they are and therefore `get_instance` is thread safe. The code is also exception safe i.e. it will be behave as expected if $\text{get}_{\text{instance}}$ throws an exception.

Another approach is:

```
static Singleton* getSingletonInstance()
{
    static Singleton instance;
    return &instance;
}
```

In C++11, the above is guaranteed to perform thread-safe initialisation.

Before C++11, the typical way to solve this issue was double-checked locking.

3.4 Strategy (B)

Strategy defines a family of algorithms, encapsulating each one, and making them interchangeable. It lets the algorithm vary independently from clients that use it and that too at run-time.

```
#include <iostream>
#include <memory>

using namespace std;

class Sort {
  public:
    virtual void sort() const = 0;
};

class CountingSort : public Sort {
  public:
    void sort() const override {
        cout << "Performing counting sort...\n";
    }
};

class HeapSort : public Sort {
    void sort() const override {
        cout << "Performing heap sort...\n";
    }
};

class MergeSort : public Sort {
    void sort() const override {
        cout << "Performing merge sort...\n";
    }
};

/// Fancy algorithm which uses sort as part of the algorithm
/// It can switch sorting technique at runtime
class FancyAlgorithm {
  public:
    FancyAlgorithm(Sort* sort)
        : m_sort(unique_ptr<Sort>(sort)) {}
```

```
    void run() const {
        cout << "Running algorithm...\n";
        m_sort->sort();
    }

  private:
    const unique_ptr<Sort> m_sort;
};

int main() {
    FancyAlgorithm algo(new CountingSort());
    algo.run();
    return 0;
}
```

```
Running    algorithm...
Performing counting    sort...
```

3.5 Observer (B)

The observer pattern is used to allow an object to publish/broadcast/push changes to its state. Other objects subscribe to be immediately notified of any changes.

Note that we can have a middle layer which manages memory of the app (subject) and devices (observers). Currently, we populate everything on the stack and main() function "owns" that memory.

```
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

class Device {
  public:
    virtual void update() = 0;
    virtual ~Device() {}
};

class IPhone : public Device {
```

```
  public:
    void update() override {
        cout << "Updating iPhone...\n";
    }
};

class IPad : public Device {
  public:
    void update() override {
        cout << "Updating iPad...\n";
    }
};

class AndroidPhone : public Device {
  public:
    void update() override {
        cout << "Updating android phone...\n";
    }
};

class App {
  public:
    /// subscribe === attach === register
    virtual void subscribe(Device&) = 0;
    /// unsubscribe === detach === unregister
    virtual void unsubscribe(Device&) = 0;
    /// push === notify
    virtual void push() = 0;
    virtual ~App() {}
};

class FancyApp : public App {
  public:
    FancyApp()
        : m_subscribed_devices() {}
    void subscribe(Device& device) override {
        m_subscribed_devices.push_back(&device);
    }
    void unsubscribe(Device& device) override {
        /// since order of devices in the vector doesn't
```

```
        /// matter for this application, we can use a simple
        /// O(1) remove from vector technique IF we know
        /// the index in the vector. After that, we can swap
        /// item at index with the last element and then
        /// delete last element. Not doing it now though
        /// since we don't store indices.
        for (auto it = m_subscribed_devices.begin();
             it != m_subscribed_devices.end();)
            if (*it == &device)
                it = m_subscribed_devices.erase(it);
            else
                ++it;
    }
    void push() override {
        for (auto device : m_subscribed_devices)
            device->update();
    }

  private:
    vector<Device*> m_subscribed_devices = {};
};

int main() {
    FancyApp app;
    IPhone iphone;
    IPad ipad;
    AndroidPhone android_phone;

    app.subscribe(iphone);
    app.subscribe(ipad);
    app.subscribe(android_phone);

    cout << "First push\n";
    app.push();
    cout << "\n";

    app.unsubscribe(ipad);
    cout << "Second push\n";
    app.push();
```

```
    return 0;
}
```

```
First       push
Updating    iPhone...
Updating    iPad...
Updating    android     phone...

Second      push
Updating    iPhone...
Updating    android     phone...
```

Note that:

- Currently, push is one-way i.e. from app (subject) to devices (observers)
  - Typically, in observer pattern, the observer has a reference back to the subject. That is because of 2-way communication when oberver's `update()` function gets the state from the subject, thereby forcing the need of subject.
  - In the example above, we are just printing stuff so don't need the subject's state.

3.6 Factory Method (C)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Also the determination of what object to create can be made at run-time due to polymorphic type interface.

```
#include <cassert>
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

class Type {
  public:
    virtual ~Type(){};
```

```
    virtual void print() const = 0;
};

class IntegerType : public Type {
  public:
    void print() const override {
        cout << "I am an integer type\n";
    }
};

class StringType : public Type {
  public:
    void print() const override {
        cout << "I am a string type\n";
    }
};

class EnumType : public Type {
  public:
    void print() const override {
        cout << "I am an enum type\n";
    }
};

enum class TypeKind { INTEGER, STRING, ENUM };

class TypeFactory {
  public:
    virtual ~TypeFactory() {}
    virtual unique_ptr<Type> make_type(TypeKind) const = 0;
};

class MyHWTypeFactory : public TypeFactory {
  public:
    /// as we can see in the function implementation,
    /// we need to know the concrete type in order to
    /// create it; that's why we have a dotted arrow
    /// (depends on or uses but doesn't have a reference)
    /// in this case, depends on is because of "creates" relationship
    /// from concrete type factory to concrete object being
```

```
    /// created in the UML.
    unique_ptr<Type>
    make_type(const TypeKind type_kind) const override {
        switch (type_kind) {
            case TypeKind::INTEGER:
                return make_unique<IntegerType>();
            case TypeKind::STRING:
                return make_unique<StringType>();
            case TypeKind::ENUM:
                return make_unique<EnumType>();
            default:
                assert(false);
        }
    }
};

int main() {
    MyHWTypeFactory tfac;
    auto t1 = tfac.make_type(TypeKind::STRING);
    t1->print();
    auto t2 = tfac.make_type(TypeKind::ENUM);
    t2->print();
    auto t3 = tfac.make_type(TypeKind::INTEGER);
    t3->print();

    return 0;
}
```

```
I am a string type
I am an enum type
I am an integer type
```

In this implementation, we always create a brand *new* type object. Instead, if needed, we can have a `getOrMake` type method which only creates a new object if one doesn't exists, else returns the old one. This can be good (depending on requirements) because all clients going through the factory will then have access to one type and hence clients wont have duplicate copies lying around.

3.7 Abstract Factory (C)

Similar to factory method but instead of creating one product, we can create a set/collection of related products. This is helpful for instance we need platform specific stuff e.g. for GUI applications, it doesn't make sense to get a mac button with a windows toolbar, where button and toolbar are objects. Instead, we can have MacFactory and WindowsFactory where both implement factory interface which has getButton and getToolbar methods. If the client gets a MacFactory, they can not get incompatible versions of products i.e. one for mac and one for windows. Good discussion on differences between factory and abstract factory here.

3.8 Builder (C)

Builder is a sort of like a superset of factory method in the sense that it encapsulates creation of objects. However, builder pattern solves a problem one layer above just the creation of objects. It is used to create complex objects with constituent parts that must be created in some specifc order or using a specific algorithm. An external class controls the construction algorithm. This implies that builder pattern allows for the dynamic creation of objects based upon easily interchangeable algorithms. It also implies that this pattern separates the construction of a complex object from its representation so that the same c onstruction process can create different representations.

Here's a good discussion on difference between factory and builder patterns.

Here's an example of builder pattern:

```
#include <cassert>
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

class Robot {
  public:
    virtual ~Robot(){};
    virtual void set_head(const string&) = 0;
    virtual void set_body(const string&) = 0;
    virtual void set_legs(const string&) = 0;
```

```
    virtual void print() const = 0;
};

class MyRobot : public Robot {
  public:
    void set_head(const string& head) override {
        m_head = head;
    }
    void set_body(const string& body) override {
        m_body = body;
    }
    void set_legs(const string& legs) override {
        m_legs = legs;
    }

    void print() const override {
        cout << "I am a robot with:\n";
        cout << "head: " << m_head << "\n";
        cout << "body: " << m_body << "\n";
        cout << "legs: " << m_legs << "\n";
    }

  private:
    string m_head;
    string m_body;
    string m_legs;
};

/// Builder who just builds specific parts
class RobotBuilder {
  public:
    virtual ~RobotBuilder() {}
    virtual void build_head() = 0;
    virtual void build_body() = 0;
    virtual void build_legs() = 0;
    virtual shared_ptr<Robot> get_robot() = 0;
};

class OldRobotBuilder : public RobotBuilder {
  public:
```

```
    /// Notice that old robot builder here allocates
    /// the actual robot objects, therefore we have
    /// to provide the explicit concrete robot type.
    /// This is the "constructs" arrow from concrete
    /// builder to concrete product in UML diagrams.
    /// We don't have to own the robot, but in this
    /// design we do.
    OldRobotBuilder()
        : m_robot(make_shared<MyRobot>()) {}

    void build_head() override {
        m_robot->set_head("Tin head");
    }
    void build_body() override {
        m_robot->set_body("Tin body");
    }
    void build_legs() override {
        m_robot->set_legs("Tin legs");
    }

    shared_ptr<Robot> get_robot() override {
        return m_robot;
    }

  private:
    shared_ptr<Robot> const m_robot;
};

class NewRobotBuilder : public RobotBuilder {
  public:
    /// see ctor comment in OldRobotBuilder
    NewRobotBuilder()
        : m_robot(make_shared<MyRobot>()) {}

    void build_head() override {
        m_robot->set_head("Aluminium head");
    }
    void build_body() override {
        m_robot->set_body("Aluminium body");
    }
```

```
    void build_legs() override {
        m_robot->set_legs("Aluminium legs");
    }

    shared_ptr<Robot> get_robot() override {
        return m_robot;
    }

  private:
    shared_ptr<Robot> const m_robot;
};

/// Director who lays out how the robot is built
/// e.g. in what order parts are assembled.
class RobotEngineer {
  public:
    RobotEngineer(shared_ptr<RobotBuilder> robot_builder)
        : m_builder(robot_builder) {}

    void construct() {
        m_builder->build_head();
        m_builder->build_body();
        m_builder->build_legs();
    }

    void set_builder(shared_ptr<RobotBuilder> builder) {
        m_builder = builder;
    }

    shared_ptr<Robot> get_robot() {
        return m_builder->get_robot();
    }

  private:
    shared_ptr<RobotBuilder> m_builder;
};

int main() {
    RobotEngineer director(make_shared<OldRobotBuilder>());
    director.construct();
```

```
    director.get_robot()->print();
    cout << "\n";
    director.set_builder(make_shared<NewRobotBuilder>());
    director.construct();
    director.get_robot()->print();
    return 0;
}
```

```
I       am          a       robot   with:
head:   Tin         head
body:   Tin         body
legs:   Tin         legs

I       am          a       robot   with:
head:   Aluminium   head
body:   Aluminium   body
legs:   Aluminium   legs
```

Note: Builder and director are following a class should be responsible for one thing only. Director lays out the plan for building, and delegates the actual building to builder whose job is to just build individual parts. This way, director has abstracted out the building. Note: We could've omitted director and let builder build the whole product as well. However, this way, we can have multiple directors (plans) and multiple builders (old builder, new builder) and we can pick and choose combinations at run-time while not disturbing one another. Also, this way we need N + M implementations rather than N * M, where N = number of directors and M = number of builders.

3.9 Adaptor (S)

Adapter pattern lets class with different interfaces to work together by creating a common object by which they may communicate and interact.

```
#include <cassert>
#include <iostream>
#include <memory>
#include <vector>

using namespace std;
```

```
/// Adaptee
class NewAPIClass {
  public:
    void new_api() const {
        cout << "Calling new api...\n";
    }
};

/// Adapter interface
class Adapter {
  public:
    virtual ~Adapter() {}
    virtual void legacy_api() const = 0;
};

/// Adapter concrete interface
class LegacyToNewAPIAdapter : public Adapter {
  public:
    void legacy_api() const override {
        // redirect to new api
        // at this point, we can "massage"
        // as we desire. For example, we can pass
        // extra/less arguments to the new api if
        // needed
        return adaptee.new_api();
    }

  private:
    // Concrete adapter can either own adapteee
    // or have a reference/pointer to it
    NewAPIClass adaptee;
};

int main() {
    const auto adapter =
        make_unique<LegacyToNewAPIAdapter>();
    // client code which expects legacy_api
    // but we route it to new_api using adapter
    adapter->legacy_api();
    return 0;
```

```
}

Calling new api...
```

3.10 Facade (S)

Imagine a subsystem consisting of lots of class with lots of complex interactions via their interfaces. Now from an outsider point of view, we want to perform one or more set of common tasks on that system. Facade pattern provides a simple unified interface to perform those common tasks. The facade then abstracts/hides out all the complexity of the subsystem within it while the clients just use the simple unified interface since that's all they care about rather than the internals/guts of the subsystem. Therefore, facade provides a higher level interface simultaneously decoupling the client from the complex subsystem.

The difference between facade and builder is that in builder, we abstracted out the creation of complex objects, while in facade, we abtracted out the complex interfaces of the subsystem.

Once you understand the motivation of facade, it's pretty simple to implement. Here's an example taken from here:

```
// The example we consider here is a case of a customer
// applying for mortgage
// The bank has to go through various checks to see if
// Mortgage can be approved for the customer
// The facade class goes through all these checks and
// returns if the morgage is approved

#include <iostream>
#include <string>

using namespace std;

// Customer class
class Customer {
  public:
    Customer(const string& name)
        : name_(name) {}
    const string& Name(void) {
        return name_;
```

```
    }

  private:
    Customer(); // not allowed
    string name_;
};

// The 'Subsystem ClassA' class
class Bank {
  public:
    bool HasSufficientSavings(Customer c, int amount) {
        cout << "Check bank for " << c.Name() << endl;
        return true;
    }
};

// The 'Subsystem ClassB' class
class Credit {
  public:
    bool HasGoodCredit(Customer c, int amount) {
        cout << "Check credit for " << c.Name() << endl;
        return true;
    }
};

// The 'Subsystem ClassC' class
class Loan {
  public:
    bool HasGoodCredit(Customer c, int amount) {
        cout << "Check loans for " << c.Name() << endl;
        return true;
    }
};

// The 'Facade' class
class Mortgage {
  public:
    bool IsEligible(Customer cust, int amount) {
        cout << cust.Name() << " applies for a loan for $"
             << amount << endl;
```

```
        bool eligible = true;

        eligible = bank_.HasSufficientSavings(cust, amount);

        if (eligible)
            eligible = loan_.HasGoodCredit(cust, amount);

        if (eligible)
            eligible = credit_.HasGoodCredit(cust, amount);

        return eligible;
    }

  private:
    Bank bank_;
    Loan loan_;
    Credit credit_;
};

// The Main method
int main() {
    Mortgage mortgage;
    Customer customer("Morty");

    bool eligible = mortgage.IsEligible(customer, 1500000);

    cout << "\n"
         << customer.Name() << " has been "
         << (eligible ? "Approved" : "Rejected") << endl;

    return 0;
}
```

```
Morty applies for a loan for $1500000
Check bank for Morty
Check loans for Morty
Check credit for Morty

Morty has been Approved
```

3.11 Proxy (S)

The proxy pattern is used to provide a surrogate or placeholder object, which references an underyling actual object i.e. object which we proxy to. The proxy provides the same public interface as the underyling object, adding a level of indirection by accepting requests from a client and passing these two the real object as necessary.

Proxy pattern is a good example of delegation.

Below another good explanation from here. It explains how we can get object to perform operations lazily.

The Proxy pattern is used when you need to represent a complex object by a simpler one. If creating an object is expensive in time or computer resources, Proxy allows you to postpone this creation until you need the actual object. A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

There are several cases where a Proxy can be useful:

1. If an object, such as a large image, takes a long time to load.
2. If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
3. If the object has limited access rights, the proxy can validate the access permissions for that user. Proxies can also be used to distinguish between requesting an instance of an object and the actual need to access it. For example, program initialization may set up a number of objects which may not all be used right away. In that case, the proxy can load the real object only when it is needed.

Following is an example from the same reference as above:

```
// Proxy is part of Structural Patterns
// Structural Patterns deal with decoupling the interface
// and implementation of classes and objects
// A Proxy provides a surrogate or placeholder for another
// object to control access to it.

// The example we consider here a math class
// The proxy provides an interface but the real class is
// only initiated when it is used
#include <iostream>
```

```cpp
#include <string>

using namespace std;

// The 'Subject interface
class IMath {
  public:
    virtual double Add(double x, double y) = 0;
    virtual double Sub(double x, double y) = 0;
    virtual double Mul(double x, double y) = 0;
    virtual double Div(double x, double y) = 0;
};

// The 'RealSubject' class
class Math : public IMath {
  public:
    double Add(double x, double y) {
        return x + y;
    }
    double Sub(double x, double y) {
        return x - y;
    }
    double Mul(double x, double y) {
        return x * y;
    }
    double Div(double x, double y) {
        return x / y;
    }
};

// The 'Proxy Object' class
class MathProxy : public IMath {
  public:
    MathProxy() {
        math_ = nullptr;
    }
    virtual ~MathProxy() {
        if (math_)
            delete math_;
    }
```

```
    double Add(double x, double y) {
        return getMathInstance()->Add(x, y);
    }
    double Sub(double x, double y) {
        return getMathInstance()->Sub(x, y);
    }
    double Mul(double x, double y) {
        return getMathInstance()->Mul(x, y);
    }
    double Div(double x, double y) {
        return getMathInstance()->Div(x, y);
    }

  private:
    Math* math_;
    Math* getMathInstance(void) {
        if (!math_)
            math_ = new Math();
        return math_;
    }
};

// The Main method
int main() {
    // Create math proxy
    MathProxy proxy;

    // Do the math
    cout << "4 + 2 = " << proxy.Add(4, 2) << endl;
    cout << "4 - 2 = " << proxy.Sub(4, 2) << endl;
    cout << "4 * 2 = " << proxy.Mul(4, 2) << endl;
    cout << "4 / 2 = " << proxy.Div(4, 2) << endl;

    return 0;
}
```

```
4 + 2 = 6
4 - 2 = 2
4 * 2 = 8
4 / 2 = 2
```

3.12 Bridge (S)

Visitor, Iterator (like in STL), and now bridge, solve one common problem: reducing combinatoric explosion of implementations. If we have `M` class1 and `N` class2, then we will need `M * N` implementations if class1 and class2 depend on each other. The key is that we abstract out class1 and class2 separately, thereby reducing combinations of implementations we have to implement to `M + N`. Having said that, we still have `M * N` possible combinations that our system represents. Using patterns like bridge, visitor, and iterator (STL), get away with writing `M + N` classes to represent `M * N` combinations in the system. This obviously is really useful if M and N are large.

This comes up in different ways in different patterns i.e. class1 and class2 represent different things in different patterns.

In Bridge pattern, class1 and class2 are concrete abstractors and concrete implementors (see bridge UML). If we add a new class1 abstractor, we just create 1 new class for class1 and 1 new class for corresponding class2 implementor.

This video explains this concept (see first few minutes).

```
#include <cassert>
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

/// Design a spotify UI which shows
/// long form and short form (and possibly more views)
/// The views can show snippets, images, urls (and possibly
/// more)

/// Implementor in UML
class Resource {
  public:
    virtual ~Resource() {}
    virtual string get_snippet() const = 0;
    virtual string get_image() const = 0;
    virtual string get_url() const = 0;
};
```

```
/// Concrete implementor in UML
class Artist : public Resource {
  public:
    string get_snippet() const override {
        return "Artist snippet";
    }
    string get_image() const override {
        return "Artist image";
    }
    string get_url() const override {
        return "Artist url";
    }
};

class Book : public Resource {
  public:
    string get_snippet() const override {
        return "Book snippet";
    }
    string get_image() const override {
        return "Book image";
    }
    string get_url() const override {
        return "Book url";
    }
};

/// Abstraction in UML
class View {
  public:
    virtual ~View() {}
    View(Resource* resource)
        : m_resource(resource) {}
    virtual void show() const = 0;

  protected:
    unique_ptr<Resource> m_resource;
};

/// Concrete abstractor in UML
```

```
class LongFormView : public View {
  public:
    LongFormView(Resource* resource)
        : View(resource) {}

    /// Long form displays image, snippet, and url
    void show() const override {
        cout << m_resource->get_image() << "\n";
        cout << m_resource->get_snippet() << "\n";
        cout << m_resource->get_url() << "\n";
    }
};

class ShortFormView : public View {
  public:
    ShortFormView(Resource* resource)
        : View(resource) {}
    /// Short form displays only image
    void show() const override {
        cout << m_resource->get_image() << "\n";
    }
};

int main() {
    /// # Views = M, # Resources = N
    /// Classes we implemented = O(M + N)
    /// Combinations we can represent = O(M * N)
    /// Classes we would have had to implement
    /// if not for bridge pattern: O(M * N)
    LongFormView long_form_book(new Book());
    LongFormView long_form_artist(new Artist());
    ShortFormView short_form_book(new Book());
    ShortFormView short_form_artist(new Artist());

    long_form_book.show();
    cout << "---\n";
    long_form_artist.show();
    cout << "---\n";
    short_form_book.show();
    cout << "---\n";
```

```
    short_form_artist.show();
    cout << "---\n";

    return 0;
}
```

```
Book    image
Book    snippet
Book    url
—
Artist  image
Artist  snippet
Artist  url
—
Book    image
—
Artist  image
—
```

I've found that bridge pattern isn't explained that well and the original description of "Decouple an abstraction from its implementation so that the two can vary independently" is confusing. A good mental model is to think this way: bridge pattern reduces # implementation to O(M + N) instead of O(M * N) where M = # of platform independent stuff and N = # platform dependent stuff". In this case, M is # of Views (`= Abstraction) and N is # of Resources (=` Implementor).

It's also good to note that ISP (Interface Segregation Principle) is not broken here. ISP says that parent interface should not have a method foo() just because one child needs it but another doesn't. In other words, all methods in the parent interface must be applicable to child classes. In the example above, short form view does not need URL but that's okay, because view "has a" resource, not "is a" resource.

Another good example is that of JVM provided in Dzone ref card cheat sheet.

Finally, we call bridge pattern a bridge pattern because the concrete abstraction (e.g. LongFormView) "bridges over" concrete implementor (e.g. Artist) but doesn't directly depend on it i.e. we can change `get_url()` in `Artist` without changing `show()` in `LongFormView`.