Object Oriented Design Concepts

Agenda

Basic Object Oriented Principles

SOLID Principles

GRASP Patterns

Object

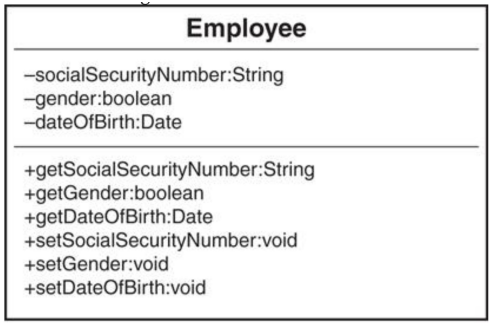
Object has a name.

It has a state (set of attr/data stored inside object)

It has well defined set of behaviors that operate on that state.

Objects expose behavior and hide data - Encapsulated, so that it can hide complexity.

Classes are templates for objects.



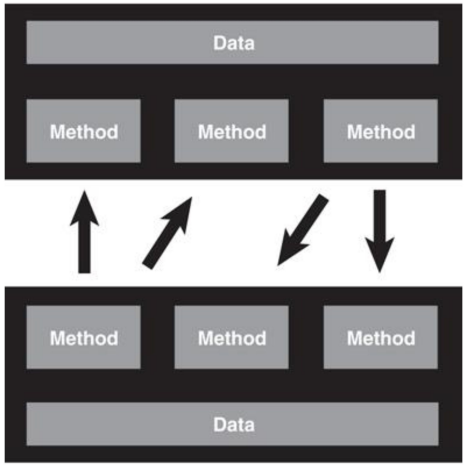
Message passing

Set of cooperating objects pass messages among themselves
Objects communicate to one another by sending messages and by Passing Data to each other

Student.SetID(ID) :
Student- Object
SetID - Message
Data - ID

The messages are of two types

- Report the current state of the object (getter)
- Perform operations on their data (setters) and change the state of the object



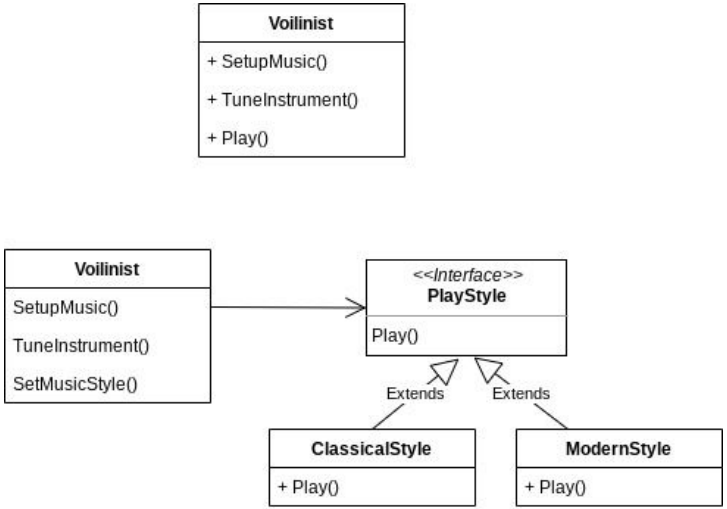
Encapsulation

Encapsulate things in your design that are likely to change.

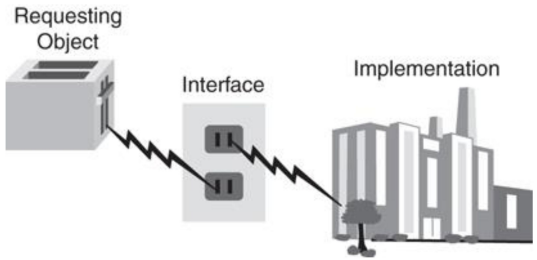
Encapsulation is not only Data hiding.

Keep volatility behind abstract interface:
Separate out stable components/classes from things which keep changing

AKA Protected Variation Pattern from GRASP



Code to an interface not to an Impl



Depend on abstraction (Abstract Base Classes in C++) not on Implementation.

- No variable should hold a pointer or reference to a concrete class.
- No class should derive from a concrete class

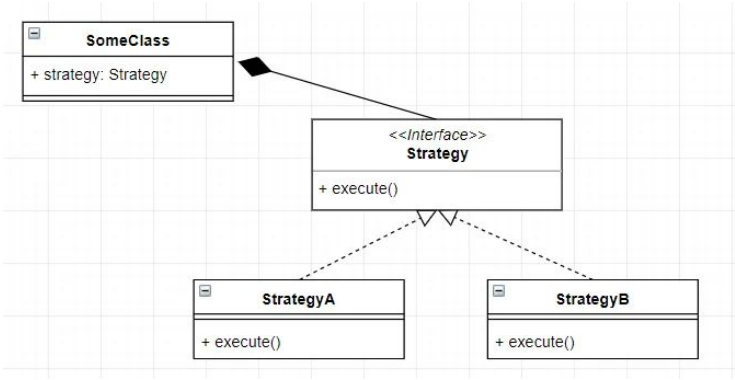
When you code to an Interface, program becomes easier to extend and modify.
Program will work with all the Interface's sub-classes seamlessly.

Change in the Impl should not change the Clients code.

Favour Composition over Inheritance

Don't use inheritance just to get code reuse

Composition provides more flexibility in design
Provides an easy way of testing using Mock objects
Using Composition + Strategy Pattern it is easier to modify the behavior at runtime



The Don’t Repeat Yourself

Avoid duplicate code

Whenever you find common behavior many places(>2):
Abstract that behavior into a class and then reuse that behavior in the common concrete classes.

Also as a general rule don’t exceed a function length >40 lines.
There will be always be oppurtunity for reuse in smaller modules/functions.

SOLID Principles

Introduced by Robert C. Martin (Uncle Bob)
From his paper *Design Principles and Design Patterns (2000)*

Five design principles which are subset of many design principles intended to make software designs more understandable, flexible and maintainable.

Single Responsibility Principle
Open and Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

The Single Responsibility Principle

Avoid GOD objects : single “know-it-all” object.

Every object in your system should have a single responsibility
All the object’s services should be focused on carrying out that responsibility

Another way of saying this is that a cohesive class does one thing well and doesn’t try to do anything else.
This implies that higher cohesion is better.

It also means that each class in your program should have only one reason to change.

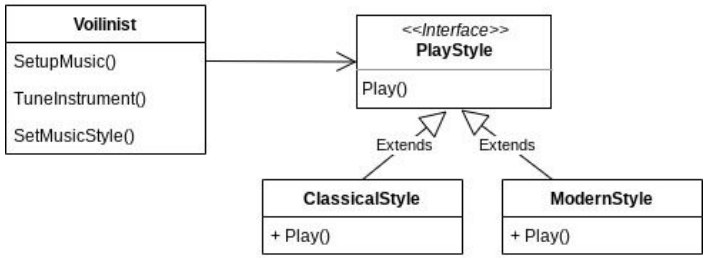
The Open-Closed Principle

1988 - Bertrand Meyer - Object Oriented Software Construction

Software artifacts (Classes/Modules) should be open for extension and closed for modification.
Open, because we will not know **every service** which client may need in future.
Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module **should not be affected** by the introduction of new services they do not need.

Abstraction, Information hiding : Don't expose private data/implementation over interface
Separate out stable components/classes from things which keep getting modified.

If the OCP is applied well, then further changes of that kind are achieved by adding new code, Not by changing old code that already works.

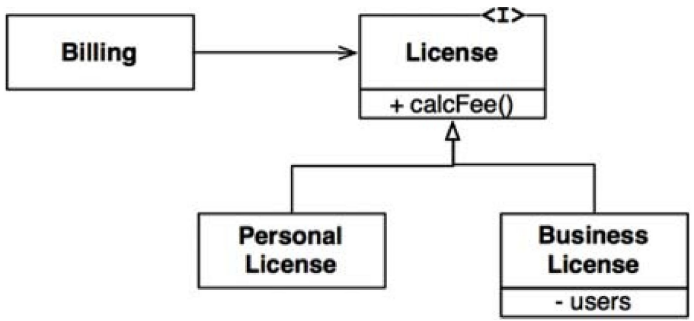


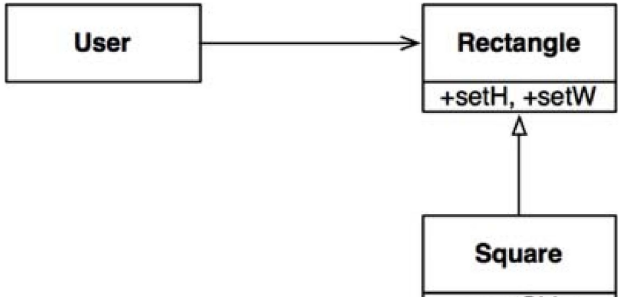
The Liskov Substitution Principle

Barbara Liskov - 1988

Subtypes must be substitutable for their base types.

Inheritance should be well designed and well behaved.





```
void Square::SetWidth(double width) {
Rectangle::SetWidth(width);
Rectangle::SetHeight(width);
}
```

```
void Square::SetHeight(double height) {
  Rectangle::SetWidth(height);
   Rectangle::SetWidth(height);
 }
```

```
void modify(Rectangle& rect) {
rect.SetWidth(5);
rect.SetHeight(4);
assert(rect.Area() == 20);
}
```

```
Rectangle rect;
modify(rect); // works fine for Rect
Square sq;
modify(sq); //wrong
```

Square is not substitutable type for Rectangle and violates LSP

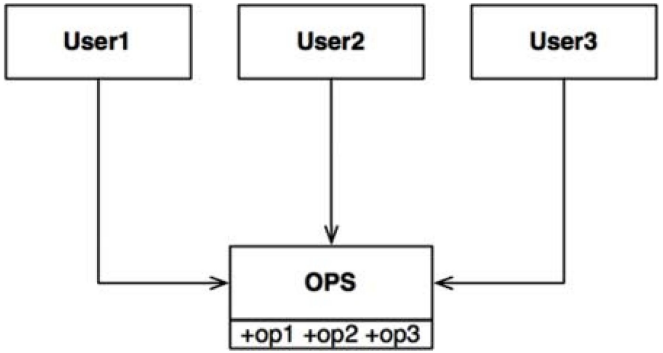
The Interface Segregation Principle

Clients shouldn’t be forced to depend on interfaces they don’t use

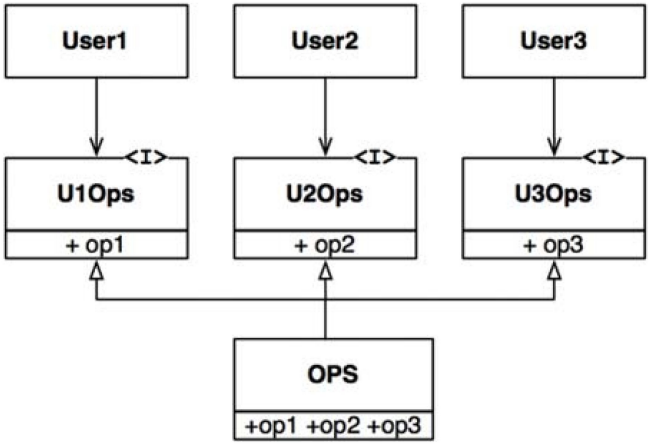
Interface Pollution/ Fat Interface
By providing too many services an Interface is less Cohesive

Clients need to implement and depend on all the interfaces/methods
They don’t need to depend on.

When one client forces a change on the fat class,
all the other clients are affected.
Thus, clients should only have to depend on methods that they actually call



Divide bloated interface into cohesive and smaller Interfaces.
New classes can implement only the interface they need.



Divided DDVProxy into smaller Interfaces

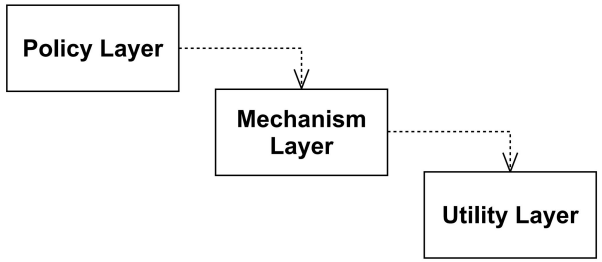
AlertListener
SummaryInfoReader
TemperatureReader
MemoryReader

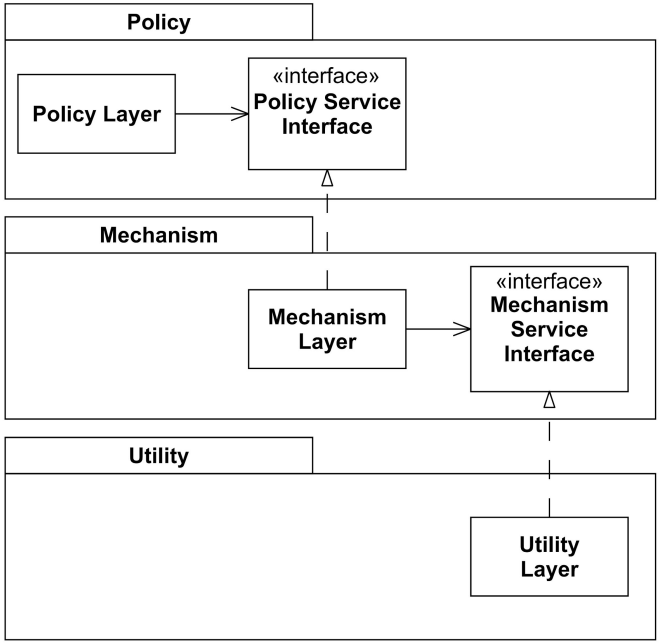
This breaks the dependence of the clients on methods that they don’t invoke,
and it allows the clients to be independent of each other.

The Dependency Inversion Principle

Don’t depend on concrete classes; depend on abstractions

High level modules should not depend on low level modules, both should depend on abstraction.
Also, abstraction should not depend on details, details should depend on abstractions.





GRASP Patterns

GRASP - General **Responsibility Assignment** Software Patterns

Craig Larman - Applying UML and Patterns

1. Information Expert - How responsibilities are assigned to an Object
2. Creator - Who creates an instance of class
3. Controller
4. Low coupling
5. High Cohesion - Single responsibility improves cohesion
6. Indirection
7. Polymorphism - Derived classes can be referenced through base classes
8. Protected variation
9. Pure fabrication

Information expert

Problem: What is a basic principle by which to assign responsibilities to objects?

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

In following example Customer class has references to all customer Orders so it is natural candidate to take responsibility of calculating total value of orders:

```
class Customer  {
        private List<Order> _orders;
        public GetOrdersTotal(Guid orderId) {     return caluculate_sum_of_orders; }
}
```

If we do not have the data we need, we would try fulfil the responsibility.

Creator

Problem: Who creates object A?

Solution: Assign class B the responsibility to create object A
if one of these is true (more is better)
– B contains or compositely aggregates A
– B records A
– B closely uses A
– B has the initializing data for A

```
class Customer  {
        private List<Order> _orders;  …..
}
```

Without customer there is no order.
So individual Order Object should be created by Customer.

The Principle of Least Knowledge

PLK Also known as Law of Demeter

A module should not know about the innards of the objects it manipulates.

Idea of loose coupling.
Objects that interact should be loosely coupled with each other.

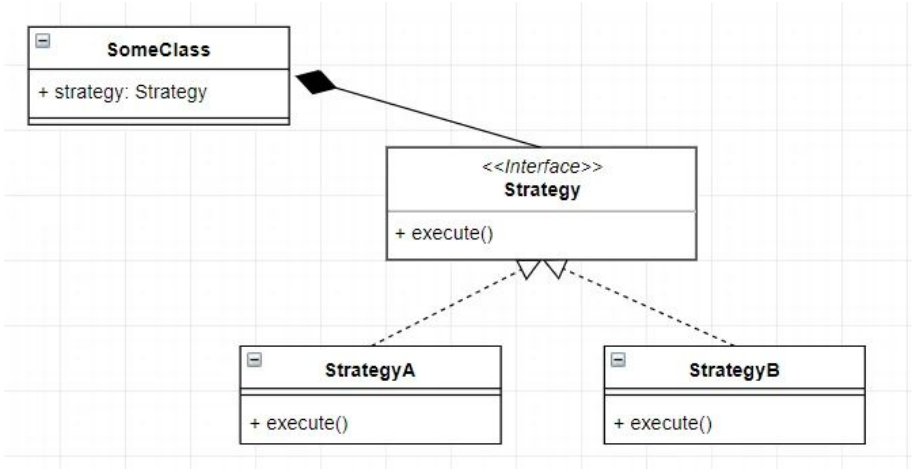
This means that an object should not expose its internal structure through accessors because to do so is to expose its internal structure (against encapsulation).
Talk only to your immediate friends.

Polymorphism

Polymorphism is fundamental principle of Object-Oriented Design.

Many popular GOF patterns depend on Polymorphism : Adapter, Command, Composite, Proxy, State, and Strategy.

```
Rest* restObj = GetCommunicator();
//make_unique<RestCurl>();
//make_unique<RestDPSL>();
```

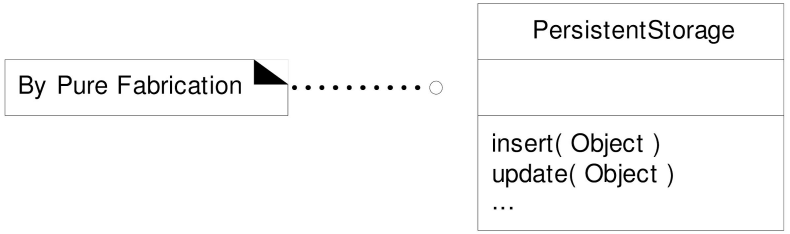


Pure fabrication

Problem: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling but solutions offered by other principles are not appropriate?
Solution: Assign a highly cohesive set of responsibilities to an **artificial or convenience class** that does not represent a problem domain concept.

For eg. we need to save **Sale** instances in a DB.
By Information Expert responsibility should go to Sale class itself, because the sale has the data that needs to be saved.

But consider the following issues:
1. Sale class need to support lot of DB related operations, none related to the concept of SALE, so the Sale class becomes incohesive.
2. Sale class has to be coupled to the DB interface so its coupling goes up.
3. Saving objects in a DB is a very general task which many other classes need support. Placing these responsibilities in the Sale class suggests there is going lots of duplication in other classes.



PersistentStorage imaginary class not represented by any real domain object.

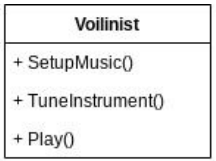
This Pure Fabrication solves the following design problems:

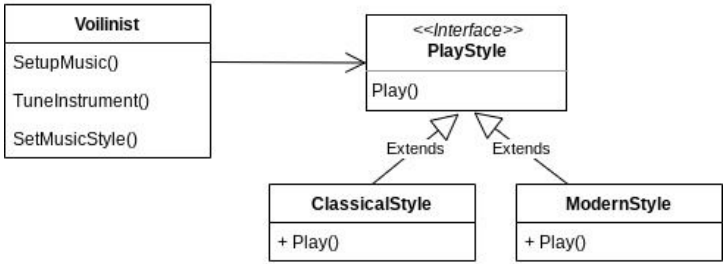
- Sale class remains well-designed, with high cohesion and low coupling.
- ThePersistentStorage class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium.
- ThePersistentStorage class is a very generic and reusable object

Protected variation

Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

Provides feasibility to insulate against future requirement changes.





References

Matt Weisfeld - The Object-Oriented Thought Process
Robert C. Martin - Clean Architecture, Agile Software Development
John F. Dooley - Software Development, Design and Coding
Craig Larman - Applying UML and Patterns
Bertrand Meyer - Object Oriented Software Construction

http://aviadezra.blogspot.com/2009/05/uml-association-aggregation-composition.html
https://wiki.c2.com/?EncapsulationIsNotInformationHiding
https://stackoverflow.com/questions/14352106/difference-between-pure-fabrication-and-indirection