# OOD Principles

## SOLID

### Single responsibility principle

### Open-closed principle

* Classes should *open for extension*, but *closed for modification*.
* Use causions to choose areas of code for potential extention.

### Liskov Substitution Principle

Derived class can substitute base class

### Interface-segregation principles

### Dependence-inversion principle

## Other guidelines

### Law of Demeter/Principle of least knowledge

Composition over inheritance

### 

# Strategy

## Problem

**base\_class**

**derived\_class1**

**derived\_class2**

**derived\_classN**

…...

**Goal**: add a number of methods method\_1(), method\_2(), … method\_K() to base\_class.

* Each method may have multiple implementations
* Choice of Impl. may or maynot be relevant to derived-classes & may change at runtime

## Alternative designs

**Single-inheritance + override**

+method1()

**derived\_class1**

**derived\_class2**

**derived\_classN**

…...

**base\_class**

+method1()

+method1()

Issue: redundant code in overrided method1 in derived classes

**Multi-level single-inheritance**

+method1()

**derived\_class1**

**derived\_class2**

**derived\_classN**

…...

**auxilary\_class2**

+method1()

**auxilary\_class1**

**base\_class**

+method1()

Issue: # inheritance increase expotentially with #functions of more than 1 impl.

**Interface / Multi-inheritance**

**base\_class**

**base\_interface1**

**deriv\_interface\_1\_1**

**deriv\_interface\_2\_1**

**base\_interface2**

**deriv\_interface\_1\_2**

**deriv\_interface\_2\_1**

**derived\_class2**

**derived\_class1**

* (older) java interface does not support implementation
* does not support changing implementation at run-time

## The Strategy design pattern

*Philosophy:* ***composition over inheritance***

context\_class

strategy\_1

impl1\_1

impl2\_1

...

...

strategy\_N

impl1\_N

impl2\_N

...

base\_class in above diagram

The base classes strategy\_k corresponds to method\_k

The derived classes impl\_k\_j corresponds to j-th implementation of method k

**Key features:** (1) ***decoupled from derived classes of context\_class*** (2) ***run-time change***

## Similar patterns (TODO)

**Template method**

**traits**

## Issues

* Increase number of objects
* Impl. exposed to clients (context\_class objects) to some-level (i.e. impl. binded to obj rather than classes)
* Overhead for communication between strategy and contex classes

Toy implementation of strategy pattern:

* context\_class contains two independent methods: method1() and method2()
* each method has two implementations
* Implementations can be chosen at initialization using bool (true: 1st, false: 2nd)
* Implementations can be changed at run time

|  |
| --- |
| #include <iostream>  #include <memory>  using namespace std;  class context\_class {  // strategies  struct strategy\_1 { virtual void operator()()=0; };  struct impl\_1\_1 : strategy\_1 {  void operator()(){ cout << "method 1: implementation 1" << endl; };  };  struct impl\_2\_1 : strategy\_1 {  void operator()(){ cout << "method 1: implementation 2" << endl; };  };  struct strategy\_2 { virtual void operator()()=0; };  struct impl\_1\_2 : strategy\_2 {  void operator()(){ cout << "method 2: implementation 1" << endl; };  };  struct impl\_2\_2 : strategy\_2 {  void operator()(){ cout << "method 2: implementation 2" << endl; };  };  // bool -> derived class mapping  static unique\_ptr<strategy\_1> get\_derived\_1 (bool flag) {  return unique\_ptr<strategy\_1>(  flag? static\_cast<strategy\_1 \*>(new impl\_1\_1)  : static\_cast<strategy\_1 \*>(new impl\_2\_1)  );  };  static unique\_ptr<strategy\_2> get\_derived\_2 (bool flag) {  return unique\_ptr<strategy\_2>(  flag? static\_cast<strategy\_2 \*>(new impl\_1\_2)  : static\_cast<strategy\_2 \*>(new impl\_2\_2)  );  };  unique\_ptr<strategy\_1> s1; // strategy object for method 1  unique\_ptr<strategy\_2> s2; // strategy object for method 2  public:  context\_class() = delete;  context\_class(bool flag1, bool flag2)  : s1(get\_derived\_1(flag1)), s2(get\_derived\_2(flag2)) {};  void set\_method1(bool flag) { s1 = get\_derived\_1(flag); };  void set\_method2(bool flag) { s2 = get\_derived\_2(flag); };  void method1() {(\*s1)();}  void method2() {(\*s2)();}  }; |

Testing:

|  |
| --- |
| int main() {  //test simple factory method  context\_class context(false, false);  context.method1(); //prints "method 1: implementation 2"  context.method2(); //prints "method 2: implementation 2"  context.set\_method1(true);  //test run-time change of strategy  context.method1(); //prints "method 1: implementation 1"  context.set\_method2(true);  context.method2(); //prints "method 2: implementation 1"  context.set\_method1(false);  context.method1(); //prints "method 1: implementation 2"  } |

# 

# Observer

## The publish-subscribe problem

subject\_object

observer\_object1

observer\_object2

observer\_object3

Observer objects are notified by the subject object when the latter changes state.

## The observer pattern

**base\_subject**

**base\_observer**

+attach(base\_observer o)

+detach(base\_observer o)

+notify()

+update()

**deriv\_subject**

**deriv\_observer**

+get\_state()

+set\_state()

-state

+ update()

- state

for (auto &o : observers) {

o->update();

}

state = subject->get\_state();

return state;

observers

subject

* The state of the subject can be pushed to all observers by notify()
* Alternatively they can be pulled by observers using get\_state()
* Observers can add/remove themselves from base\_subject at run time
* Subject /observers are unaware of each other's type, and they can each inherit from other types

**Key features:** (1) ***subjects broadcast to observers*** (2) ***subject-observer coupling is loose and dynamic***

## Java observable class

base\_subject class: class Observable

* setChanged(): signify a state change

base\_observer class: interface Observer

**Push:**

* subject.setChanged();
* observer.notifyObservers(Observable o, Object arg);

**Pull:**

* subject.setChanged();
* subject.notifyObservers(null);
* observers query subject state when they'd like too

**Issue with Java observable class**

* **Observable is a class**:since Java does not support multi-inheritance, the deriv\_subject cannot extend from another class. There's also no way to override certain implementations.
* **Inheritance from observable is forced**:sinces methods like setChanged() method is protected. So there's no way to composite Observable with user class, violating the composite over inheritance principle

## Related pattern(TODO)

**Command**

**MVC**

Toy implementation of observer pattern:

* at runtime, subscribe/unsubscribe observers (as function objects) to the subject
* observer may be the base class or the derived class, which the subject is unaware of
* once subject's member value changes, the function objects are called

|  |
| --- |
| #include <iostream>  #include <functional>  #include <unordered\_map>  #include <memory>  #include <atomic>  #include <mutex>  using namespace std;  class subject {  private:  struct observer {  public:  observer(uint64\_t i, function<void (subject &)> f) : id(i), func(f) {}  virtual void update(subject &sub) { func(sub); }    private:  uint64\_t id;  std::function<void (subject &)> func;  };    class observerI : public observer {  public:  observerI(uint64\_t i, std::function<void (subject &)> f) : observer(i, f){};  void update(subject &sub) {  cout << "(derived observer)";  func(sub);  }  };    uint64\_t next\_id;  atomic<int> value;  unordered\_map<uint64\_t, unique\_ptr<observer>> observers;  mutex m\_observer, m\_id, m\_value;  void notify() {  lock\_guard<mutex> lck(m\_observer);  for (auto &item : observers) item.second->update(ref(\*this));  }    public:  subject() : next\_id(0){};  uint64\_t add\_observer(function<void (subject &)> f, bool deriv = false) {  lock\_guard<mutex> lck\_ob(m\_observer), lck\_id(m\_id);  uint64\_t id = ++next\_id;  observers.emplace(id,  unique\_ptr<observer>(deriv? new observerI(id, f) : new observer(id, f))  );  return id;  }    void remove\_observer(uint64\_t id) {  lock\_guard<mutex> lck(m\_observer);  observers.erase(id);  }    int get\_value() {  return value;  }  void set\_value(int val){  lock\_guard<mutex> lck(m\_value);  if (val == value) return;  value = val;  notify();  }  }; |

Testing:

|  |
| --- |
| struct context {  void notified(subject &s) const{  cout << "object observes subject value" << s.get\_value() << endl;  }  };  class sub\_context : public subject{  public:  void notified(subject &s) {  int v = s.get\_value();  cout << "sub\_context object observes subject value " << v << endl;  subject::set\_value(v);  }  };  void foo(subject &s) {  cout << "function observes subject value" << s.get\_value() << endl;  }  int main(int argc, const char \* argv[])  {  subject sub;  // test 1: update value without subscribers  sub.set\_value(1);  //prints nothing    // test 2: add a lambda function subscriber, then update value  uint64\_t id1 = sub.add\_observer([] (subject &s){  cout << "lambda function observes subject value "<<s.get\_value()<<endl;  });  sub.set\_value(2);  //prints "lambda function observes subject value 2"    // test 3: add class member function subscriber, then update value  context context\_obj;  uint64\_t id2 = sub.add\_observer([context\_obj] (subject &s) {  context\_obj.notified(s);  });  sub.set\_value(3);  //prints "lambda function observes subject value 3"  //prints "object observes subject value 3"  // test 4: add a function that creates a derived observer  // also remove the class member function and a non-existing id  uint64\_t id3 = sub.add\_observer(foo, true);  sub.remove\_observer(id2);  sub.remove\_observer(100);  sub.set\_value(4);  //prints "lambda function observes subject value 4"  //prints "(derived observer)function observes subject value 4"  // test 5: add an observer who is also a subject  // add an observer to the secondary subject  sub\_context sub2;  sub\_context\* psub2 = &sub2;  uint64\_t id4 = sub.add\_observer([psub2](subject &s) {psub2->notified(s);});  uint64\_t id5 = sub2.add\_observer([](subject &s) {  cout << "observer of sub\_context observes subject value " << s.get\_value() << endl;  });  sub.set\_value(5);  //prints "sub\_context object observes subject value 5"  //prints "observer of sub\_context observes subject value 5"  //prints "(derived observer)function observes subject value 5"  //prints "lambda function observes subject value 5"  return 0;  } |

# Decorator

## Problem

**level3\_class**

+method()

**level2\_class**

+method()

**level1\_class**

+method()

void method() {

do\_something\_else();

level2\_class::method();

do\_something\_else2();

}

Design a family of classes

* all classes share the same interface
* the method of a class may adds certain steps (which we call "***condiment***") to another

## Alternative design

**base\_class**

+method()

+do\_pre\_step1()

+do\_pre\_step2()

…

+do\_post\_step1()

+do\_post\_step2()

...

void method() {

if (do\_pre\_step1()) pre\_step1();

if (do\_pre\_step2()) pre\_step2();

….

if (do\_post\_step1()) post\_step1();

if (do\_post\_step2()) post\_step2();

}

**class\_I**

+method()

+do\_pre\_step1()

+do\_pre\_step2()

…

+do\_post\_step1()

+do\_post\_step2()

...

**class\_II**

+method()

+do\_pre\_step1()

+do\_pre\_step2()

…

+do\_post\_step1()

+do\_post\_step2()

...

**class\_II**

+method()

+do\_pre\_step1()

+do\_pre\_step2()

…

+do\_post\_step1()

+do\_post\_step2()

...

Issue:

* require changging the code with new condiment
* condiment hard-wired to subclasses
* does not impose constraint on condiment dependency

## The Decorator Pattern

**Philosophy:** (1) ***open-closed principle*** (2)***composition over inheritance***(note here additional steps are composited with base-class behavior via runtime decoration, inheritance is used to enforce same supertype)

**base**

**component**

**derived component**

**base**

**decorator**

**derived**

**decorator A**

**derived**

**decorator B**

* Decorator extends the base component, thus you can pass around a decorated object in place of the original
* You can apply decoration multiple times
* Decorator adds its own behavior before/after delegating the decorated object to do the rest of job
* Decoration can be done at runtime

**Key features:**(1) ***add steps to a defined interface without modifying the original***   
(2) ***runtime, recursive decoration***

## Issues

* Order of added steps matter
* Code cannot rely on concerete component type but such types are transparent to clients
* Hard to track down bugs
* Complicates the amount of code to initialize a (potentially) decorated object (see factory and builder patterns)

## Real-world applications:

* append log to start and finsh of method
* JavaIO: filterInputStream -> bufferedInputStream -> lineNumberInputStream

## Related patterns:

Adaptor, facade, proxy

# Singleton

## Problem:

Create a "standalone" class allowing at most one instantiation

* Thus its constructor cannot be called from outside
* The instance may only be created when needed (lazy initialization)
* Outside code can get a handle to the single instance
* Life-span of the instance is for the duration of the application

The use of singleton comes more from convention than from design principles

## Alternatives:

**global variables**

* Singleton does not pollute the namespace as much
* Singleton supports lazy initialization

**static class**

* If the class is self-contained and does not depend on complex initialization, static class is a good replacement of singleton

## Static v.s. Lazy intialization

**Implementation with eager(static) initialization**

|  |
| --- |
| class singleton {  private:  singleton() {};  static shared\_ptr<singleton> unique\_instance = shared\_ptr<singleton>(new singleton());  public:  static shared\_ptr<singleton> get\_instance() { return unique\_instance; };  }; |

**Implementation with lazy initialization**

Method: double checked locking

|  |
| --- |
| class singleton {  private:  singleton() {};  static volatile shared\_ptr<singleton> unique\_instance;  public:  static shared\_ptr<singleton> get\_instance() {  if (unique\_instance == nullptr) {  synchronized (Singleton.class) { //TODO: this line is java code  if (unique\_instance == nullptr) {  unique\_instance = shared\_ptr<singleton>(new singleton());  }  }  }  return unique\_instance;  };  }; |

## subclassing a singleton

* Change constructor to protected/public
* Add registry in initialization to make sure the corresponding instance is returned
* This is typically discouraged as it is not a real singleton

## applications:

* thread pool, memory management, registry
* In general, we should spare the use of singleton

multiple class loaders can cause problem

# Factory

## Factory Method

Use parameters to determine which derived class to generate

## Abstract Factory

* create families of dependent objects
* create the overall object by calling each dependent objects

# 

# 

# Command

**client**

**invoker**

**command**

**derived\_command**

**receiver**

+action()

+execute()

+undo

+set\_command()

+execute()

+undo

Defines a binding between the invoker of request and the request handler

* client creates a derived\_command and sends it to receiver
* invoker calls the command at some point, involking derived\_command::execute()
* derived\_command::execute() calls receiver::action()

# 

Applications: log/transactions/hardware api

# Adaptor

## Problem

Two classes cannot work together because incompatible interface

**client code**

**target\_interface**

+request()

target\_interface t;

t.request();

**adaptee\_class**

+request1()

Codebase I

Downloaded library

So we need an adapter class to convert the interface of a class into the interface of another

## Object adaptor

## 

**client code**

**target\_interface**

+request()

void request() {

adaptee.request1();

}

**adaptee\_class**

+request1()

Codebase I

Downloaded library

**adapter\_class**

+request()

## 

## Class adaptor

## 

**client code**

**target\_interface**

+request()

void request() {

adaptee.request1();

}

**adaptee\_class**

+request1()

Codebase I

Downloaded library

**adapter\_class**

+request()

# Facade

* Simplifies interface by unifying a large family of interfaces

The family may have multiple facades

# Template method

## Problem

An algorithm has a very structured outline

|  |
| --- |
| primitive\_op1();  primitive\_op2();  ...  primitive\_opN(); |

But the primitive ops are versatile.

## The Template Method Pattern

Philosophy: (the holywood principle) "don't call me, I'll call you"

* High-level component controls when and how to call the low-level ones
* It seems to share the spirit of "dependency inversion principle"

**abstract\_class**

**concrete\_class**

+template\_method

+primitive\_op1

+primitive\_op2

+primitive\_op1

+primitive\_op2

primitive\_op1();

primitive\_op2();

## Comparison to Strategy

* Template method slightly more lightweighted, widely used
* Template method works by inheritance rather than composition
* The interchangeable parts in template are incomplete pieces rather than a full op.

## Two implementations of primitive\_ops:

* Abstract class
* Hook: functions with default(often empty) impl., useful for optional steps

# Iterator

## Problem

**C++ style iterators**

**iterator**

**random access iterator**

**bidirectional iterator**

**forward iterator**

**input iterator**

**output iterator**

|  |  |
| --- | --- |
| all | Iterator b(a); //copy constructor  b = a; //copy assignment |
| input | a == b; a != b //equality and inequality  \*a; a->m; //dereference as rvalue |
| output | \*a = t; \*a++ = t; //dereference as rvalue, only for mutable iterator types |
| forward | Iterator a; Iterator(); //default constructor  a++; // increamenting |
| bidirectional | --a; // decrementing |
| ramdom access | a+n; n+a; a-n; a-b; // operator +/- with random access iterators or int  a<b; a>b; a>=b; a<=b; //inequality comparisons  a+=n; a-=n; //compound assignment  a[n] //offset dereference operator |

t: object a, b: iterator (of type of t) n: integer

**Java style iterators** (similar to C++ style forward iterator)

|  |
| --- |
| template <class DATATYPE>  class iterator {  bool has\_next();  DATATYPE next();  void remove(); //optional, see next page for discussion on this function  }; |

Iterator provides an interface to access an aggregate object

* sequential access
* iterator does not expose the object's underlying representation

## The iterator pattern

**Philosophy: *single responsibility***

**aggregate**

+create\_iterator

**concrete aggregate**

+create\_iterator

**iterator**

+has\_next

+next

+remove

**concrete aggregate**

+has\_next

+next

+remove

**client**

**Key features:** (1) independent of object represenation (2) sequential access

## for statement without explicitly using iterator

for (obj o : collection) // featured since Java5 and C++11

## remove() in java iterator

* remove the last element returned by next
* should be careful about concurrency
* OK if iterator does not support it, simply throw exception

## Composite Iterator

Iterator recursively over the composite pattern

e.g. Leetcode 341

|  |
| --- |
| **class NestedIterator {**  public:  **NestedIterator(vector<NestedInteger> &nestedList)** {  for (int k = nestedList.size()-1; k>=0; --k) stk.push(nestedList[k]);  }  **int next()** {  NestedInteger ni = stk.top();  stk.pop();  return ni.getInteger();  }  **bool hasNext()** {  while (!stk.empty() && !stk.top().isInteger()) {  const vector<NestedInteger> nestedList = stk.top().getList();  stk.pop();  for (int k = nestedList.size()-1; k>=0; --k) stk.push(nestedList[k]);  }  return !stk.empty();  }    private:  stack<NestedInteger> stk;  };  // This is the interface that allows for creating nested lists.  // You should not implement it, or speculate about its implementation  class NestedInteger {  public:  // Return true if this NestedInteger holds a single integer, rather than a nested list.  bool isInteger() const;  // Return the single integer that this NestedInteger holds, if it holds a single integer  // The result is undefined if this NestedInteger holds a nested list  int getInteger() const;  // Return the nested list that this NestedInteger holds, if it holds a nested list  // The result is undefined if this NestedInteger holds a single integer  const vector<NestedInteger> &getList() const;  }; |

### Null Iterator:

Two options when the composite component has nothing to iterate over

* return null from createIterator
* return an iterator which always returns false at has\_next()

|  |
| --- |
| template <class DATATYPE>  class null\_iterator : public iterator {  next() { return null; }  bool has\_next() { return false; }  }; |

### Additional considerations:

* Ordering
* Caching

# Composite

## Problem

Represents a tree hierarchy with a recursive structure and treat individual objects (e.g. o3, o4, o5) and their compositions (e.g. o1, o2) as the same.

**o1**

**o2**

**o3**

**o4**

**o5**

## The composite pattern

**Philosophy: *trades single responsibility for transparency***

**component**

**leaf**

**composite**

**Key features:** (1) tree structure (2) leaf and non-leaf treated as identical

## Real world applications:

* GUI component layout (see MVC)
* Macro command

## Issue

* violates single responsibility principle
* client may do illegal/meaning operation on components because all components are treated as the same

# State

## Problem

Allow an object to alter its behavior when its internal state changes

|  |  |  |
| --- | --- | --- |
| **state** | **interpretation** | 1  2  3  0  IDLE  OPEN  4  STOP  STOP  DOWN  UP  CLOSE  UP  DOWN |
| **0** | **idle/ prepare to move** |
| **1** | **moving up** |
| **2** | **moving down** |
| **3** | **Prepare to idle or to open door** |
| **4** | **Door opened/error** |
|  |  |

**Input codes:** UP, DOWN, STOP, OPEN, CLOSE, IDLE

## Alternative implementation

Procedural state machine

|  |
| --- |
| struct state\_machine {  enum States {IDLING, MOVING\_UP, MOVING\_DOWN, STOPPED, DOOR\_OPENED};  enum Input {UP, DOWN, STOP, OPEN, CLOSE, IDLE, NO\_RESP};  void handle\_input (Input i) {  State s;  switch (cur) {  case IDLING:  switch (i) {  case UP: s = MOVING\_UP; break;  case DOWN: s = MOVING\_DOWN; break;  default: s = STOPPED; break; //no response or illegal input  }  break;  case MOVING\_UP:  switch (i) {  case UP: s = MOVING\_UP; break;  case STOP: s = STOPEED; break;  default: s = STOPPED; break; //no response or illegal input  }  break;  //.. sdditional states  }  handle\_state(s);  }  private:  State cur = IDLING;  void go\_to\_state(State s) { cur = s };  }; |

Issue: poor readability, hard to expand

## The State Pattern

**state**

+handle\_input1

+handle\_input2

+handle\_input3

- &machine

**state1**

**state2**

**state3**

+handle\_input1

+handle\_input2

+handle\_input3

+handle\_input1

+handle\_input2

+handle\_input3

+handle\_input1

+handle\_input2

+handle\_input3

**machine**

+ request

- cur\_state

cur\_state.handle\_input1();

machine.set\_state(machine.get\_state(3));

**Key features:** (1) ecapsulate state-based behavior (2) delegate behavior to the current state

## State v.s. Strategy

* strategy configures context object with a behavior or algorithm
* state allow the context object to changes strategy in a well-defined scheme (the finite state automata)

## Issues

Additional overhead due to additional classes and abstraction

## Applications

* TCP connection
* User interaction handler in drawing

# Proxy

## Problem

- subject

client object

service

Client needs access to a member, which:

* sits in a different address space
* expensive to access
* needs controlled access
* needs reference counting, etc.

Therefore we want a proxy layer between the two so that client is unaware of the complexity of the service

- subject

client object

proxy

service

## The Proxy Design pattern

**client**

**abstract\_service**

**proxy\_service**

**actual\_service**

real\_subject

real\_subject.request()

+request()

+request()

+request()

**Key feature:** proxy adds a level of indirection to data access  
Usage of indirections

(1) ***Remote proxy:*** hides detail about accessing data in a different address space

(2) ***Virtual proxy:*** performs optimization by doing certain expensive ops on demmand

(3) ***Protection and smart reference proxy:*** performs additional house keeping

# MVC

## Problem

design the framework for an interactive program with GUI

model

controller

view

user

Frame

Subframe

**Animation of model state**

Button A

Button B

OnMouseClick(){

}

OnPaint(){

}

OnButtonAClicked(){

}

OnButtonBClicked(){

}

...

* view displays model status
* model implements all the logic
* controller handles user action (a lot of glue work)

## Involved patterns

**Observer:** Model state change notifies view & control

**Strategy:** The view delegates to the controller to handle user actions

**Composite:** the GUI components are organized in a tree-like structure

Some secondary/possible patterns:

**State:** for cleaning up the model logic

**Adapter:** for adapting a new model to an existing view/controller

## Application: Model2 for Web applications

The model: we can use existing model off the shelf

The controller: the servelet

* Receive web-browser input as HTTP request and turn into actions
* Return view to browser via JSP

# Other patterns

## Bridge

## Builder

## Chain of responsibility

## Flyweight

## Interpreter

## Mediator

## Memento

## Prototype

## Visitor