Design Patterns in C++

4-day session

Overview	Understanding VTable concepts in C++ Polymorphism Understanding Object Layout Understanding 23 GOF Design patterns Understanding Creational, Structural and Behavioural design patterns
Duration	Four days - 32 hours (8 hours a day) 50% of lecture, 50% of practical labs.
Trainer	http://www.linkedin.com/in/pravinkumarsinha
Audience	Professionals developing medium scale to large scale software
Prerequisite	**Knowledge of Object Oriented Concepts** Oops concept like Abstraction, Inheritance, Polymorphism would help.
Setup	Machine with g++ installed.

Lecture

Lecture session will be course content presentation through the trainer.
Any source code example related to the topic will be demonstrated, it would include executing the binaries.
Complete lecture material can be downloaded from
http://www.minhinc.com/training/advance-dp-slides.pdf

Labs

Labs session would be completely hands on session where each example (with example data and execution instruction) would be provided to the students. Students can verify their results with the results provided in the material.

Day 1 Morning

Lecture - C++ Object Model

- C++ Object Model
- Single Inheritance
- Multiple Inheritance

Lecture - Vtable structure

- Vtable Structure
- Virtual methods and SI, MI
- Virtual base class
- Thunk based and Offset based object model

Lecture - Design Pattern Overview

- GOF Design Pattern overview
  - Creational Design Pattern
  - Structural Design Patterns
  - Behavioural Design Patterns

Lecture - GOF Creational Design Pattern

- New Object Creation
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Builder Pattern
  - Singleton Pattern
- Cloning
  - Prototype Pattern

Day 1 Afternoon

Lab

Day 2 Morning

Lecture - GOF Structural Design Pattern

- Composition patterns
  - Adapter pattern
  - Bridge pattern
  - Facade pattern
  - Flyweight pattern
  - Proxy pattern
- Inheritance pattern
  - Proxy pattern
  - Adapter pattern
- Compostion + Inheritance patterns
  - Composite pattern
  - Decorator pattern

Day 2 Afternoon

Lab

Day 3 Morning

Lecture - GOF Behavioural Design Pattern - 1

- Behavioral design patterns
- Recursive method calls
  - Chain of responsibility pattern
  - Interpreter pattern
- Non recursive calls
  - Callback method calls
  - Command pattern
  - State pattern
  - Observer pattern
  - Visitor pattern

Day 3 Afternoon

Lab

Day 4 Morning

Lecture - GOF Behavioural Design Pattern - 2

- Non recursive calls, cont..
  - Direct Method call
  - Command (subclassed) pattern
  - Itertor pattern
  - Mediator pattern
  - Memento pattern
  - Strategy pattern
  - Template method pattern

Day 4 Afternoon

Lab

Design Patterns Essentials

Design Patterns Essenstials- Training Course

Minh, Inc.

DISCLAIMER

Text of this document is written in Bembo Std Otf(13 pt) font.

Code parts are written in Consolas (10 pts) font.

This training material is provided through **Minh, Inc.**, B'lore, India
Pdf version of this document is available at http://www.minhinc.com/training/advance-dp-slides.pdf
For suggestion(s) or complaint(s) write to us at sales@minhinc.com

Document modified on Sep-30-2019

Document contains 64 pages.

Day 1 Morning

1. C++ Object Model

- C++ Object Model
  - Single Inheritance
  - Multiple Inheritance

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_0

Day 1 Morning

1. C++ Object Model

- C++ Object Model
- Single Inheritance
- Multiple Inheritance

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_2

Day 1 Morning

1. C++ Object Model

- C++ Object Model
- Single Inheritance
- Multiple Inheritance

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_3

Day 1 Morning

2. Vtable structure

- Vtable Structure
  - Virtual methods and SI, MI
  - Virtual base class
  - Thunk based and Offset based object model

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_4

Day 1 Morning

2. Vtable structure

- Vtable Structure
- Virtual methods and SI, MI
  - Virtual base class
  - Thunk based and Offset based object model

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_5

Day 1 Morning

2. Vtable structure

- Vtable Structure
  - Virtual methods and SI, MI
- Virtual base class
  - Thunk based and Offset based object model

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_6

Day 1 Morning

2. Vtable structure

- Vtable Structure
  - Virtual methods and SI, MI
  - Virtual base class
- Thunk based and Offset based object model

Refer
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap8_7

Day 1 Morning

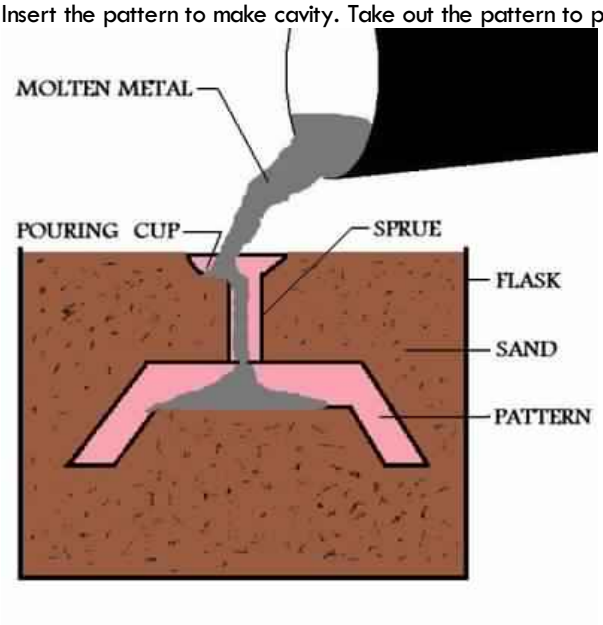
3. Design Pattern Overview

- GOF Design Pattern Overview
  - Creational Design Patterns
  - Structural Design Patterns
  - Behavioural Design Patterns
- Design Patterns and Programming language

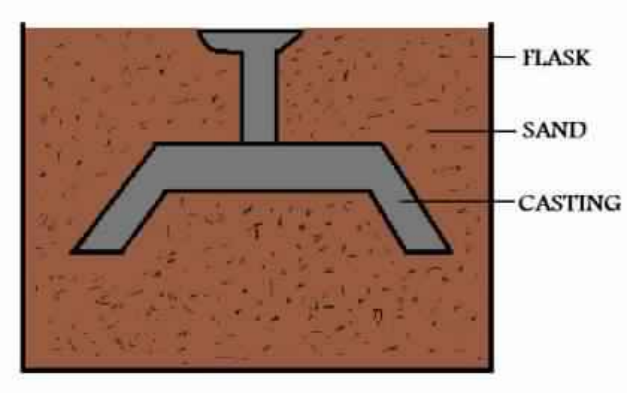
Design patterns are pre defined patterns used for
- implementing different creation scenarios
- implementing different structural scenarios
- implementing different behavioural scenarios

Design Pattern is very similar to pattern used in metal casting

Insert the pattern to make cavity. Take out the pattern to put molten metal.



Model is ready as per the pattern.



A pattern is a basic unit of designing. When analysis phase is done the closest pattern is considered for the solution or even some hybrid patterns are also checked for solution. Since patterns are well defined and tested way of solving the problem, once identified they can prove stable design and non modifiable code. The more available patterns are in different aspects of designing area supported in a language, the more powerful the language is. In the document 23 GOF design patterns are discussed and implemented through C++ code. 23 design patterns are distributed among 3 categories. Creational, structural and behavioral. This document is written in sequel and contains description for creational structural and behavioral patterns.

-Creational pattern
A class/item can be instantiated in many different scenarios. Out of which 5 scenarios are common.

- Factory method

© www.minhinc.com

Items/classes need to be created for a single platform through a static method in the Item class iteself by passing some hint (i.e. name). This way creation of the item is hidden from the calling code. Calling code recieves base/item class pointer through which it can operate through its methods, independent of which drive class object is instantiated.

- Abstract factory

Items/classes in needs to be created for different categories. Item in same category (i.e. platform) can be created through a platform specific factory. Whereas platform specific factories are extended from a generic abstract factory. Once factory is instantiated depending upon the platform, items specific to the platform can be created through the factory.

- Singleton

Only one instance of a class is permittable. More instance of a class should return the same unique instance.

- Builder

As the name suggest a builder pattern is used when an entity (builder) creates a complex object in parts (predefined). A builder pattern exposes various methods of creating complex structures and it is the director who calls for these methods and once the structure is ready the builder returns to the director.

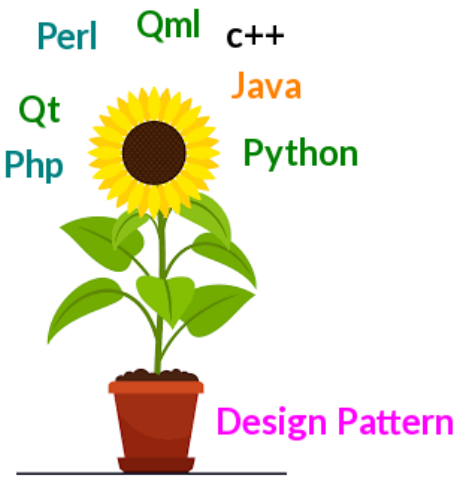
- Prototype

When cloning an object the prototype pattern is used; after an object is created the pattern modifies its state and the data structure. However, if a copy of the current state of a new object is needed cloning can be done.. Every implementation class overrides abstract class clone function.

Day 1 Morning

3. Design Pattern Overview

- GOF Design Pattern Overview
  - Creational Design Patterns
  - Structural Design Patterns
  - Behavioural Design Patterns
- Design Patterns and Programming language



All object oriented languages, i.e. c++, Java, Python, Qml, uses Design Patterns for large scale stable product design. While achieving creational, structural or behavioural complex scenarios design patterns are used to create relationship among the classes. Hybrid design patterns can also be created to achieve the gloal. So Design pattern is the entity that glow in object oriented language based products.

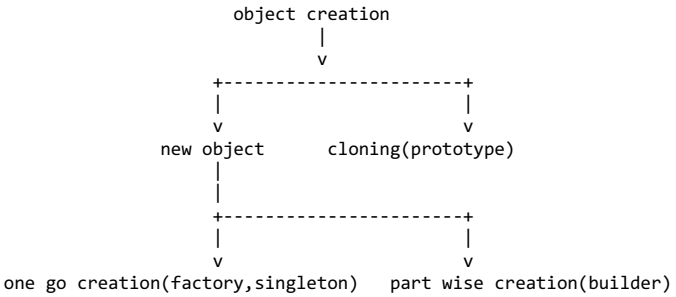
Day 1 Morning

4. GOF Creational Design Pattern

- New Object Creation
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Builder Pattern
  - Singleton Pattern
- Cloning
  - Prototype Pattern

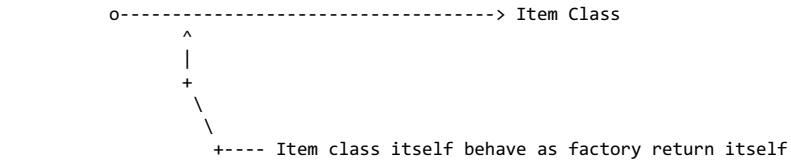
Creational design patterns

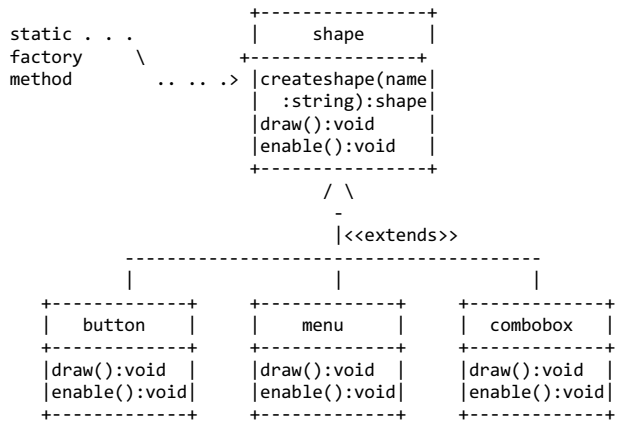
Creational design pattern deals with both creating the object and the component. A component is created or generated in two ways. Generating from scratch (first of its kind) and generating from cloning. In first method the component can be generated at ago or part by part in generic fashion. A factory class makes new component creations at ago, whereas builder makes part by part (for intricate), while prototype does cloning.



- Factory Method Pattern

Items/classes need to be created for a single platform through a static method in the Item class iteself by passing some hint (i.e. name). This way creation of the item is hidden from the calling code. Calling code recieves base/item class pointer through which it can operate through its methods, independent of which drive class object is instantiated.





Code

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class widget {
public:
 static widget* createwidget(string&);
 virtual void draw() = 0;
 virtual void enable(bool) = 0;
```

© www.minhinc.com

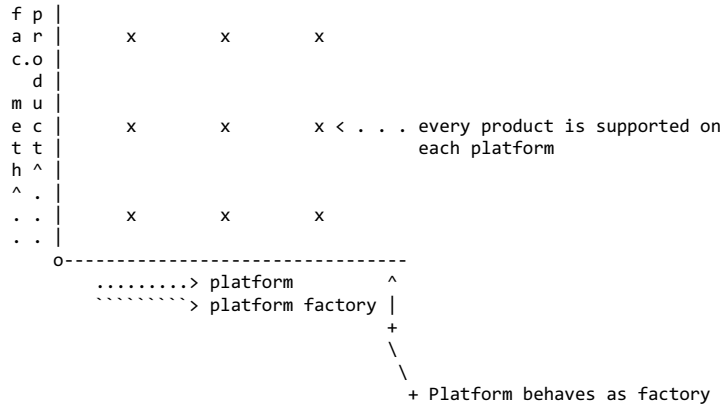
```
};

class button : public widget {
public:
 void draw() { cout << "Button::draw"<<endl; }
 void enable(bool e_d) { cout << "Button::enable "<< e_d<<endl; }
};
class menu : public widget {
public:
 void draw() { cout << "Menu::draw"<<endl; }
 void enable(bool e_d) { cout << "Menu::enable "<< e_d<<endl; }
};
class combobox : public widget {
public:
 void draw() { cout << "Combobox::draw"<<endl; }
 void enable(bool e_d) { cout << "Combobox::enable "<< e_d<<endl; }
};

widget* widget::createwidget( string& name) {
if (name == "button")
 return new button;
else if (name == "menu")
 return new menu;
else
 return new combobox;
}

int main( void ) {
 vector<widget*> shape;
 string choice;
 int i=0;
 while (true) {
  cout << "enter you choice, button, menu, combobox, e(exit)"<<endl;
  cin >> choice;
  if (choice == "e")
   break;
   shape.push_back( widget::createwidget( choice ) );
  }
  for (int i=0; i < shape.size(); i++){
   shape[i]->enable(true);
   shape[i]->draw();
  }
  for (int i=0; i < shape.size(); i++)
   delete shape[i];
 return 0;
}
```

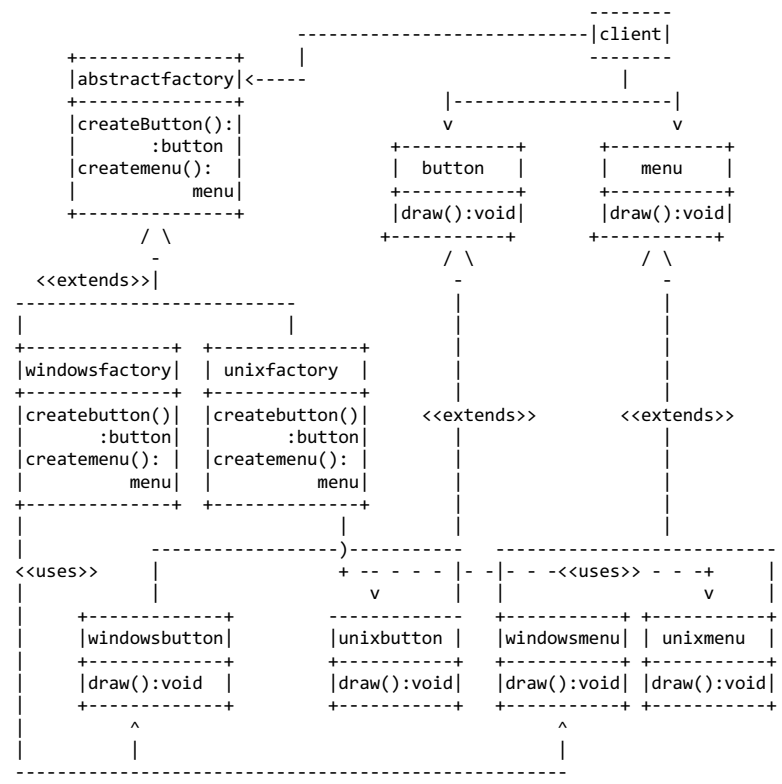
Result

```
pi@raspberrypi:~/tmp/MISC/gc $ ./a.out
enter you choice, button, menu, combobox, e(exit)
menu
enter you choice, button, menu, combobox, e(exit)
button
enter you choice, button, menu, combobox, e(exit)
combobox
enter you choice, button, menu, combobox, e(exit)
button
enter you choice, button, menu, combobox, e(exit)
e
Menu::enable 1
Menu::draw
Button::enable 1
Button::draw
Combobox::enable 1
Combobox::draw
Button::enable 1
Button::draw
```

- Abstract Factory

Items/classes in needs to be created for two different categories. Item in same category (i.e. platform) can be created through a platform specific factory. Whereas platform specific factories are extended from a generic abstract factory. Once factory is instantiated depending upon the platform, items specific to the platform can be created through the factory. So platform and actual product makes a orthogonal coordinates.






```
#include <iostream>
using namespace std;
#define WINDOWS

class widget {
public:
   virtual void draw() = 0;
};

class linuxbutton : public widget {
public:
   void draw() { cout << "linuxbutton "; }
};
class linuxmenu : public widget {
public:
   void draw() { cout << "linuxmenu "; }
};

class windowsbutton : public widget {
public:
   void draw() { cout << "WindowsButton "; }
};
class windowsmenu : public widget {
public:
   void draw() { cout << "WindowsMenu "; }
};

class factory {
public:
   virtual widget* create_button() = 0;
   virtual widget* create_menu() = 0;
};

class linuxfactory : public factory {
public:
   widget* create_button() {
      return new linuxbutton; }
   widget* create_menu()   {
      return new linuxmenu; }
};

class windowsfactory : public factory {
public:
   widget* create_button() {
      return new windowsbutton; }
   widget* create_menu()   {
      return new windowsmenu; }
};

int main( void ) {
 factory* factory;
#ifdef MOTIF
 factory = new linuxfactory;
#else // WINDOWS
```

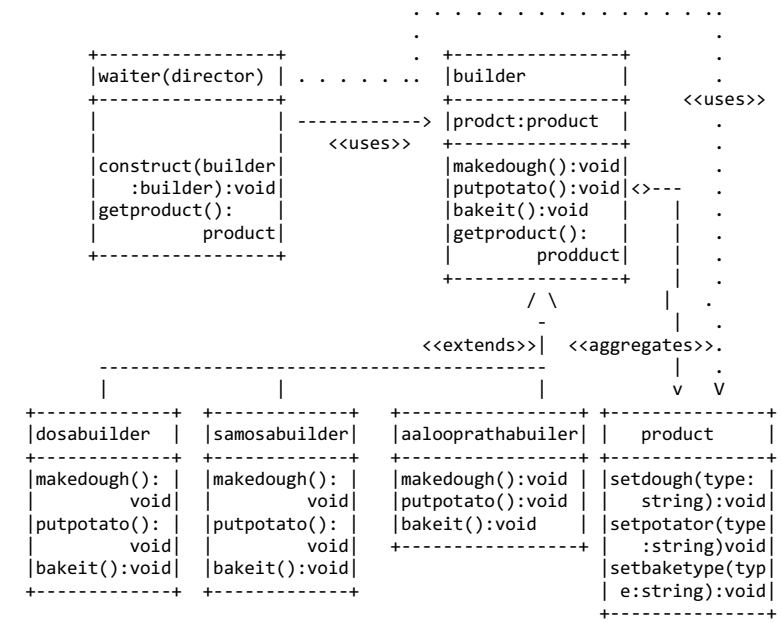
```
 factory = new windowsfactory;
#endif

 widget* w1 = factory->create_button();
 widget* w2 = factory->create_menu();
 w1->draw();
 w2->draw();
}
```

Result

```
WindowsButton
WindowsMenu
```

- Builder Pattern

As the name suggest a builder pattern is used when an entity (builder) creates a complex object in parts (predefined). A builder pattern exposes various methods of creating complex structures and it is the director who calls for these methods and once the structure is ready the builder returns to the director. In Indian restaurants there are built food items including Masala Dosa, Samosa, and Aaloo Paratha. These items are built through three different Builders The builders follow same steps, which are (a) Making dough (b) Putting potato (c) Baking it. The waiter who receives the order undertakes the three steps of builder without knowing the details. Once the food item is ready the waiter returns the item to the customer.



Code

```
#include <iostream>
using namespace std;

class product{
public:
void dough(const string& type) {
cout<<"dough : "<<type<<endl;
}
void aloo(const string& type) {
cout<<"aloo : "<<type<<endl;
}
void fry(const string& type) {
cout<<"fry : "<<type<<endl;
}
};

class builder{
public:
 virtual void makedough()=0;
 virtual void putaloo()=0;
 virtual void fry()=0;
 virtual product* getresult()=0;
};


class dosabuilder:public builder{
 product* _product;
public:
 dosabuilder(){
  _product=new product;
 }
 void makedough(){
  _product->dough("wet");
```

```
 }
 void putaloo(){
  _product->aloo("fried");
 }
 void fry(){
  _product->fry("oil fried");
 }
 product* getresult(){
  return _product;
 }
};

class samosabuilder:public builder{
 product* _product;
public:
 samosabuilder(){
  _product=new product;
 }
 void makedough(){
  _product->dough("dry");
 }
 void putaloo(){
  _product->aloo("fried");
 }
 void fry(){
  _product->fry("deep fried");
 }
 product* getresult(){
  return _product;
 }
};

class director{
public:
void construct(builder *bp){
bp->makedough();
bp->putaloo();
bp->fry();
}
};

int main(void){
director *directori=new director;
directori->construct(new dosabuilder);
directori->construct(new samosabuilder);
return 0;
}
```

Output

```
pi@raspberrypi:~/tmp/MISC/gc $ ./a.out
dough : wet
aloo : fried
fry : oil fried
dough : dry
aloo : fried
fry : deep fried
```

- Singleton pattern

As name suggests when only single object of a class is created, the pattern we use to achieve this is singleton pattern. If object creation function is used more than once the already created object is returned. i.e. Someone needs to draw many coffee mugs and all look equal but different in position on screen. So one object is enough as far as look is concern. Position can be adjusted externally. Here factory would create a singleton object. In a province where king can be one. All calls to get the king would return the same king.

```
#include <iostream>
using namespace std;

class singletonclass {
 singletonclass(){}
 static singletonclass* instance;
public:
 static singletonclass* getinstance() {
  if (! instance )
   instance = new singletonclass;
  return instance;
 }
};
singletonclass* singletonclass::instance = 0;


int main( void ) {
 singletonclass *inst1=singletonclass::getinstance();
 singletonclass *inst2=singletonclass::getinstance();
 cout<<"instances address"<<endl<<inst1<<endl<<inst2<<endl;
return 0;
}
```

Result

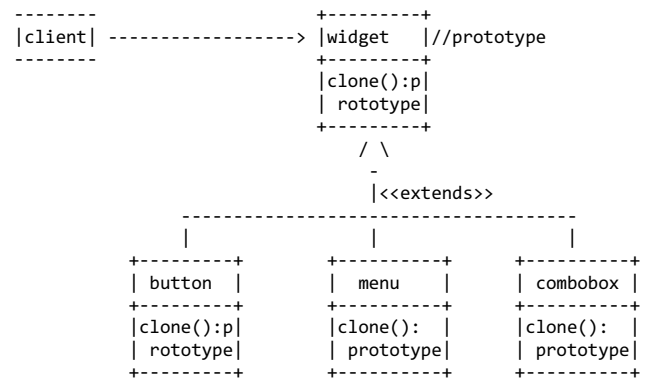
```
instances address
0x678f10
0x678f10
```

Day 1 Morning

4. GOF Creational Design Pattern

- New Object Creation
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Builder Pattern
  - Singleton Pattern
- Cloning
  - Prototype Pattern

When cloning of an object is required this pattern is used When an object is created it modifies its state and other data structure. When a new object carry requirement of having a copy of current state of an object, object has to clone itself. Every implementation class overrides abstract class clone function. a guru makes shishya(students) those are clone of him.They carry same knowledge and guru does not lose anything.



Code

```
#include <iostream>
#include <vector>
using namespace std;

class widget {
public:
 static widget* shapetypes[4];
 virtual widget* clone() = 0;
 virtual void draw() = 0;
 virtual void enable(bool) =0;
};

class button : public widget {
public:
 widget* clone(){return new button;}
 void draw() { cout << "Button::draw"<<endl; }
 void enable(bool e_d) { cout << "Button::enable "<< e_d<<endl; }
};
class menu : public widget {
public:
 widget* clone(){return new menu;}
 void draw() { cout << "Menu::draw"<<endl; }
 void enable(bool e_d) { cout << "Menu::enable "<< e_d<<endl; }
};
class combobox : public widget {
public:
 widget* clone(){return new combobox;}
 void draw() { cout << "Combobox::draw"<<endl; }
 void enable(bool e_d) { cout << "Combobox::enable "<< e_d<<endl; }
};

widget* widget::shapetypes[4]={0,new button, new menu, new combobox};


int main(void) {
 vector<widget*> shape;
 int choice;
 while (true) {
  cout << "enter you choice, (1)button, (2)menu, (3)combobox, 9(exit)"<<endl;
  cin >> choice;
  if (choice == 9)
   break;
  shape.push_back( widget::shapetypes[choice]->clone() );
 }
  for (int i=0; i < shape.size(); i++){
```

```
    shape[i]->enable(true);
    shape[i]->draw();
   }
   for (int i=0; i < shape.size(); i++)
    delete shape[i];
 return 0;
}
```

Output

```
pi@raspberrypi:~/tmp/MISC/gc $ ./a.out
enter you choice, (1)button, (2)menu, (3)combobox, 9(exit)
1
enter you choice, (1)button, (2)menu, (3)combobox, 9(exit)
2
enter you choice, (1)button, (2)menu, (3)combobox, 9(exit)
3
enter you choice, (1)button, (2)menu, (3)combobox, 9(exit)
9
Button::enable 1
Button::draw
Menu::enable 1
Menu::draw
Combobox::enable 1
Combobox::draw
```

Day 2 Morning

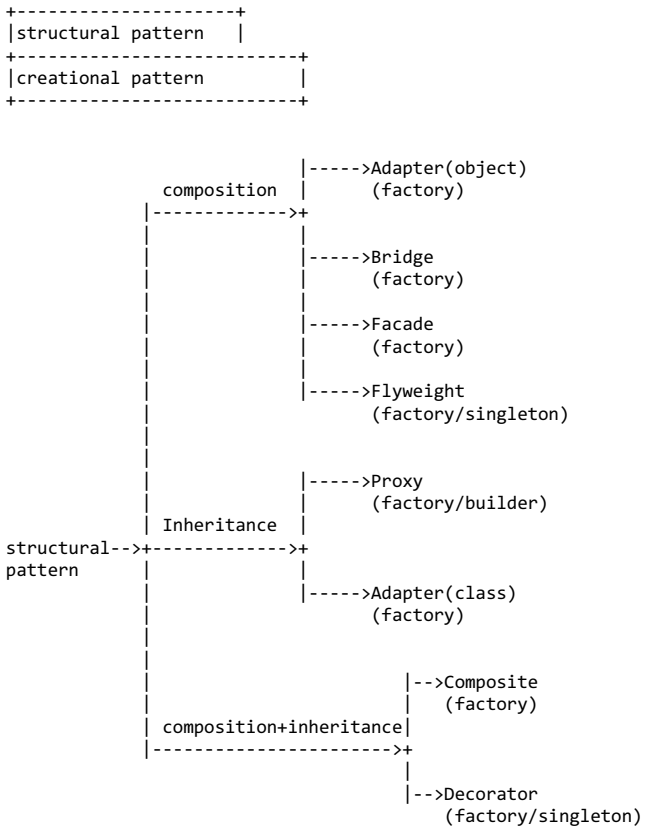
5. GOF Structural Design Pattern

- Composition patterns
  - Adapter pattern
  - Bridge pattern
  - Facade pattern
  - Flyweight pattern
  - Proxy pattern
- Inheritance pattern
  - Proxy pattern
  - Adapter pattern
- Compostion + Inheritance patterns
  - Composite pattern
  - Decorator pattern

Structural patterns deal with the layout of how the attributes of the class/design is distributed and related. The class deals with static structure of the design and not how object interacts with each other. There are various categories it can be distributed in.

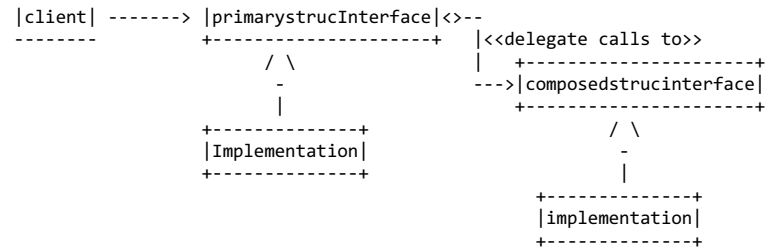
a) When a structure composes another different structure.
b) When a structure is derived from others, inherits and extends parent property.
c) When a structure composes others and inherit the same.

Adapter (object), bridge, facade, flyweight are examples of a structure that compose a different structure. Proxy and Adapter (class) are examples of structures derived from other structures. Composite and decorator derive and compose the same, apart from these other structural patterns have a base in one of the creational patterns. A structural pattern is based on a specific creational pattern. Similarly behavioral pattern has a base on structural patterns.



- Compostion Pattern

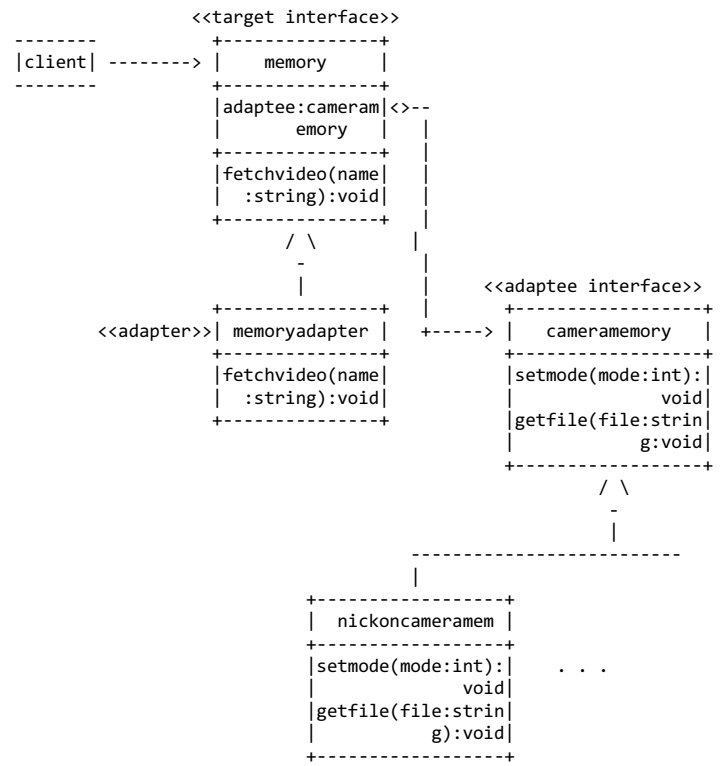
Structural patterns work through compositing other structures. The first, structure that composes other structures, somewhere delegates the client calls in order to maximize its usage. Composed structure is generally passed to the first one through the client. This kind of pattern have primary structure and composed structure where primary structure delegates the client call to composed one, both client and primary structure keeps interfaces of the next one and mix their code with interface methods. This makes client and primary structure code close for changes.



- Adpater

When adaptee's interface varies from client interface, a client expects this Pattern to be used. It changes client supported interface to adaptee's interface and hence enable the client to use adaptee. An adapter can be of two types. First type one, 'object type' where adaptee interface is contained in adapter and any subclass of adaptee can be contained in adapter at run time; whereas the second one is 'class type' where adaptee is also a base class for adapter. When two individuals communicate with one another with different languages, both have to use a multilingual two way adapter to let them communicate even though they do not understand each other's language.

object type:



Code

```
#include <iostream>
#include <string>
using namespace std;

class cameramemory {
public:
int mode;
string videofile;
virtual void setmode(int)=0;
virtual const string& getfile(const string&)=0;
};

class nikoncameramemory:public cameramemory {
public:
void setmode(int modep) {
mode=modep;
if (modep==1)
 cout<<"nikoncameramemory::CHANGED TO READ MODE ";
else
 cout<<"nikoncameramemory::CHANGED TO WRITE MODE ";
}
const string& getfile(const string& name) {
if (!mode)
 videofile="ERROR in MODE, can not read in write mode, change the mode ";
else{
```

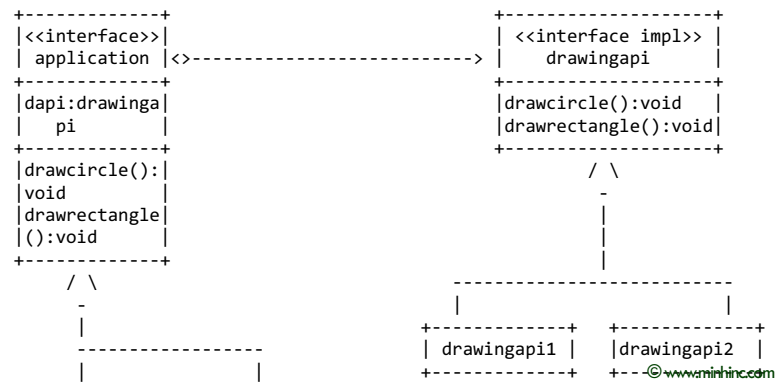
```
cout<<"nikoncameramem::serving file : MEMFILE1"<<endl;
videofile="MEMFILE1";
}
return videofile;
}
};

class memory{
public:
memory(){
adaptee = new nikoncameramemory;
}
cameramemory *adaptee;
virtual const string& fetchvideo(const string&)=0;
virtual ~memory(){
delete adaptee;
}
};

class memoryadapter:public memory{
public:
const string& fetchvideo(const string& name){
adaptee->setmode(1);
return adaptee->getfile(name);
}
};

int main(){
memory *madapter=new memoryadapter;
cout<<madapter->fetchvideo("earth song")<<endl;
delete madapter;
return 0;
}
```

Output

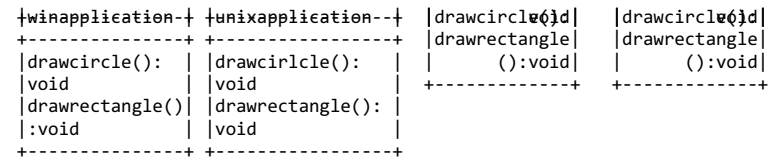
```
nikoncameramemory::CHANGED TO READ MODE
nikoncameramem::serving file : MEMFILE1
MEMFILE1
```

- Bridge Pattern

When an abstract has several implementations and each implementation (say abstract implementation) grows to a number of sub implementations then classes at sub implementation level exists in many numbers. Same set of sub implementations are available for each abstract implementation. It is important to separate abstract implementation tree with the sub implementation tree. This way abstract implementation and its sub implementation grows independently. In sub implementation tree each sub implementation creates bridge to an abstract implementation. sub implementation is interface to client where the abstract implementation is the real implementation which is delegated for clients call to interfaces. For example there are drawings, APIs used across application types. All application types inherit all drawing APIs. If there are three application types supported and one new drawing API is available then three more inheritance are required. Better way to avoid that is to separate application hierarchy from drawing API hierarchy. Each application implementation creates a bridge to API implementation interface. In a country's political system there can be multiple parties and each party has a president, vice president, secretary etc. Now the number of parties and posts increases adjacently. An individual wants to be a secretary in party A and another person wants to be a president in the same party A. If we have an abstract class Post and then for each Post implementation, all parties are subclasses, then the total number of party classes would be too many. Here we can have Posts and Parties separate hierarchy and for a person to become a Post holder in a Party, will instantiate the party implementation by passing the Post implementation.
Like new party A (new secretary).This way he becomes secretary in party A.



© www.minhinc.com



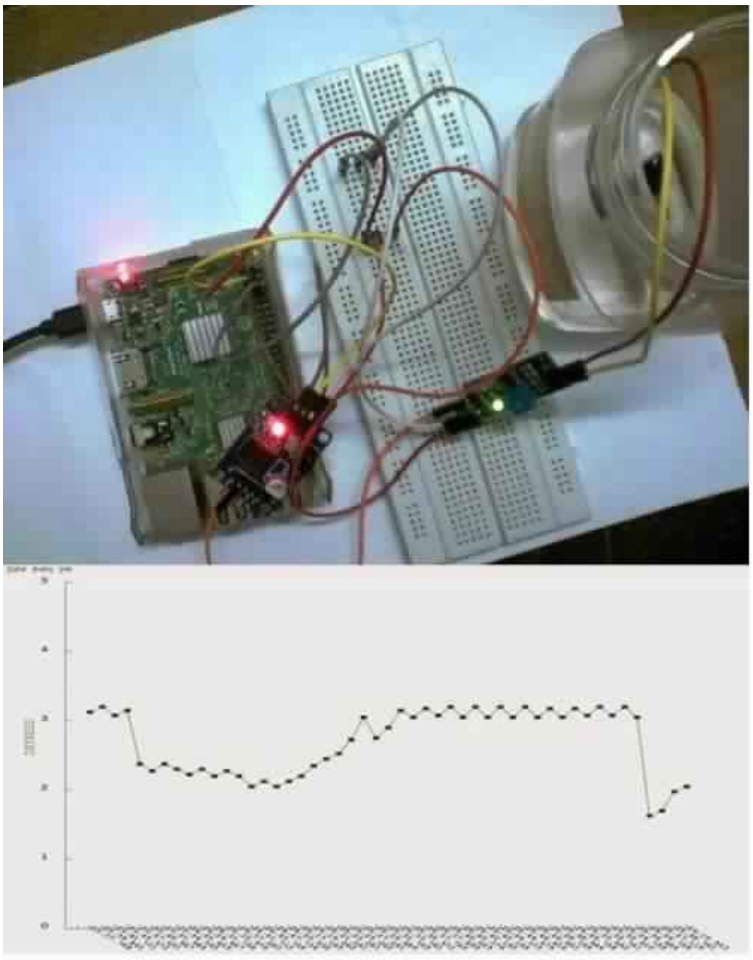

```
#include <iostream>
using namespace std;

class drawingapi {
public:
virtual void drawcircle(int,int,int)=0;
virtual void drawrectangle(int,int,int,int)=0;
};

class drawingapi1:public drawingapi{
public:
void drawcircle(int x1, int x2, int radius) {
cout<<"drawingapi1::drawcircle x1:"<<x1<<" x2:"<<x2<<" radius:"<<radius<<endl;
}
void drawrectangle(int x1, int x2, int width, int height){
cout<<"drawingapi1::drawrectangle x1:"<<x1<<" x2:"<<x2<<" width:"<<width<<" height:"<<height<<endl;
}
};

class drawingapi2:public drawingapi{
public:
void drawcircle(int x1, int x2, int radius) {
cout<<"drawingapi2::drawcircle x1:"<<x1<<" x2:"<<x2<<" radius:"<<radius<<endl;
}
void drawrectangle(int x1, int x2, int width, int height){
cout<<"drawingapi2::drawrectangle x1:"<<x1<<" x2:"<<x2<<" width:"<<width<<" height:"<<height<<endl;
}
};

class application {
public:
drawingapi* dapi;
application(drawingapi *drawingapip):dapi(drawingapip){}
virtual void drawcircle(int x,int y,int radius)=0;
virtual void drawrectangle(int x,int y,int w,int h)=0;
virtual ~application(){delete dapi;}
};

class windowsapplication:public application {
public:
using application::application;
void drawcircle(int xp,int yp,int radiusp){
dapi->drawcircle(xp,yp,radiusp);
}
void drawrectangle(int xp,int yp,int wp,int hp){
dapi->drawrectangle(xp,yp,wp,hp);
}
};

class unixapplication:public application {
public:
using application::application;
void drawcircle(int xp,int yp,int radiusp){
dapi->drawcircle(xp,yp,radiusp);
}
void drawrectangle(int xp,int yp,int wp,int hp){
dapi->drawrectangle(xp,yp,wp,hp);
}
};

int main(void){
application *winapp=new windowsapplication(new drawingapi1);
winapp->drawcircle(50,50,100);
winapp->drawrectangle(20,40,100,200);
application *unixapp=new unixapplication(new drawingapi2);
unixapp->drawcircle(25,25,100);
unixapp->drawrectangle(60,80,100,200);
delete winapp;
delete unixapp;
return 0;
}
```

Output

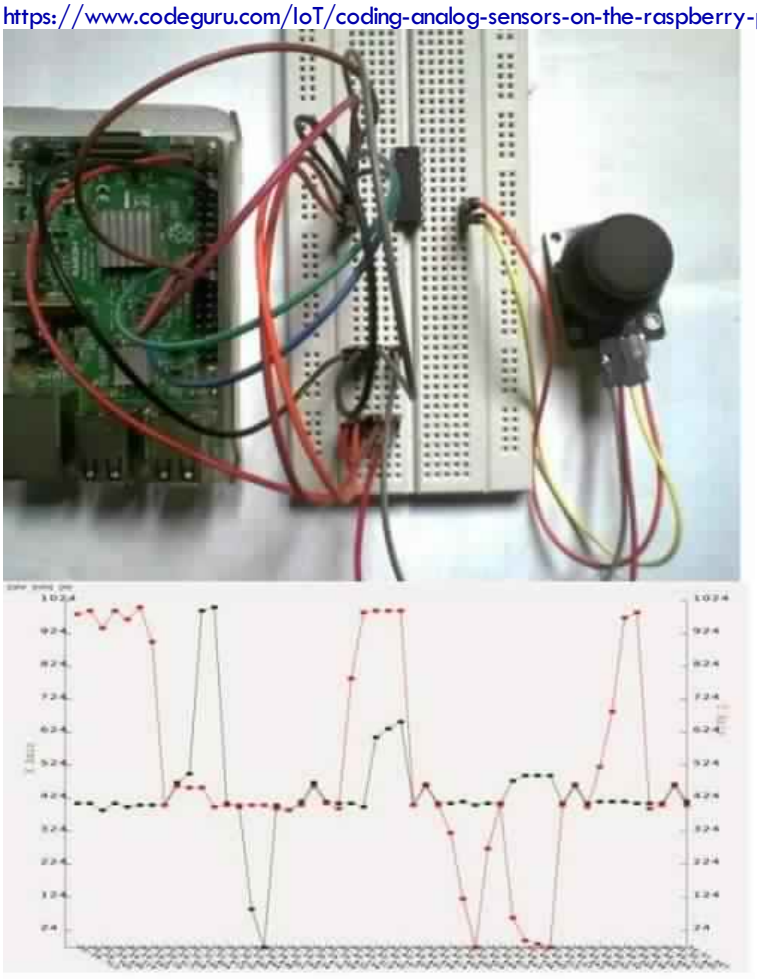
```
drawingapi1::drawcircle x1:50 x2:50 radius:100
drawingapi1::drawrectangle x1:20 x2:40 width:100 height:200
drawingapi2::drawcircle x1:25 x2:25 radius:100
drawingapi2::drawrectangle x1:60 x2:80 width:100 height:200
```

Article

https://www.codeguru.com/IoT/coding-sensors-on-the-rpi3.html

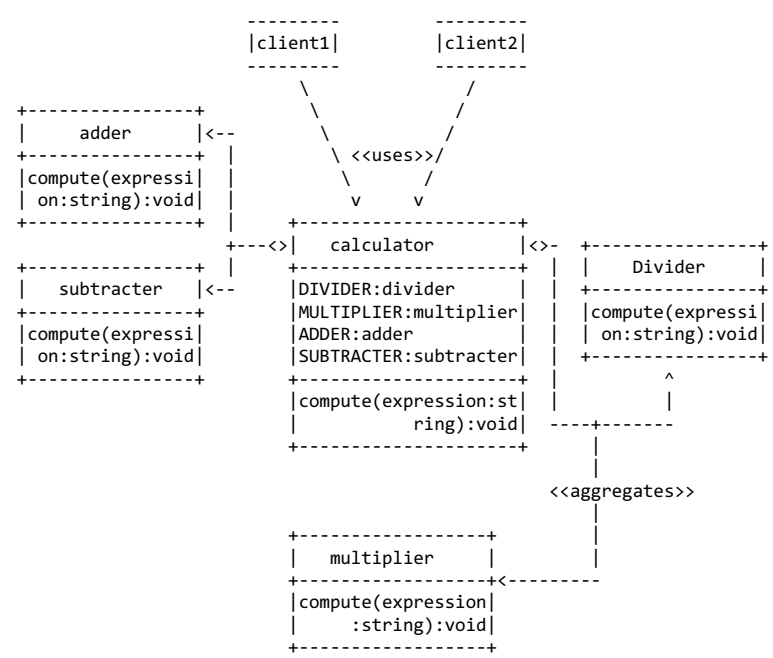


https://www.codeguru.com/IoT/coding-analog-sensors-on-the-raspberry-pi3.html



- Facade

When an extra interface is required for handling complex underlying Components, facade pattern is used. This makes various underlying classes easy to use through facade interface. Facade interface delegates calls to underlying implementations. For example there are separate components that undertakes specific tasks. An adder; adds two numbers, multiplier; multiplies two numbers and a divider divides two numbers. There is a facade abstract data type calculator which provides interface for evaluating an expression and delegating the expression parts to separate components. There are many departments in a province and ministers taking care of each department. It's the prime minister's mandate though to interact with people for their complex issues though he may not handle every department.




```
#include <iostream>
#include <regex>
using namespace std;

class adder{
public:
float compute(float leftp, float rightp){
 cout<<"adder:: left:"<<leftp<<" right:"<<rightp<<endl;
 return leftp+rightp;
}
};

class subtracter{
public:
float compute(float leftp, float rightp){
 cout<<"subtracter:: left:"<<leftp<<" right:"<<rightp<<endl;
 return leftp-rightp;
}
};

class multiplier{
public:
float compute(float leftp, float rightp){
 cout<<"multiplier:: left:"<<leftp<<" right:"<<rightp<<endl;
 return leftp*rightp;
}
};

class divider{
public:
float compute(float leftp, float rightp){
 cout<<"divider:: left:"<<leftp<<" right:"<<rightp<<endl;
 return leftp/rightp;
}
};

class calculator {
adder *adderi;
subtracter *subtracteri;
multiplier *multiplieri;
divider *divideri;
public:
float compute(const string& expression){
smatch sm;
if(regex_search(expression,sm,regex("(.+?)\\+(.*)")) == true)
 adderi->compute(this->compute(sm.str(1)),this->compute(sm.str(2)));
```

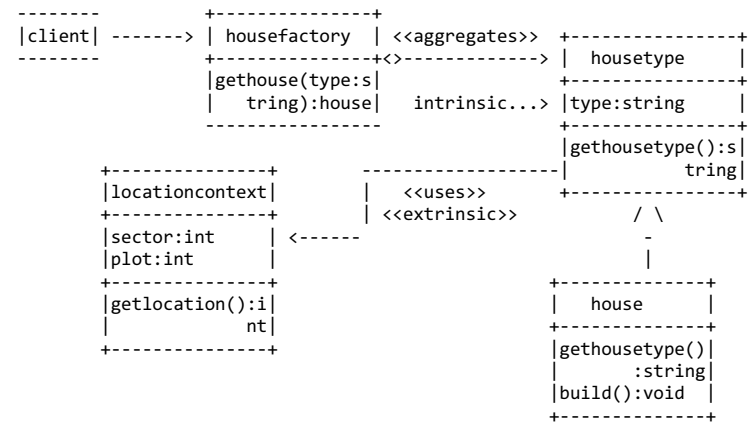
```
else if(regex_search(expression,sm,regex("(.+?)\\-(.*)")) == true)
 subtracteri->compute(this->compute(sm.str(1)),this->compute(sm.str(2)));
else if(regex_search(expression,sm,regex("(.+?)\\*(.*)")) == true)
 multiplieri->compute(this->compute(sm.str(1)),this->compute(sm.str(2)));
else if(regex_search(expression,sm,regex("(.+?)\\/(.*)")) == true)
 divideri->compute(this->compute(sm.str(1)),this->compute(sm.str(2)));
else
 return stof(expression);
}
};

int main(void){
calculator calc;
cout<<"expression:  1-2+4*3-6/2+8*3-2*70/10"<<endl;
cout<<calc.compute("1-2+4*3-6/2+8*3-2*70/10");
return 0;
}
```

Output

```
expression:  1-2+4*3-6/2+8*3-2*70/10
subtracter:: left:1 right:2
multiplier:: left:4 right:3
divider:: left:6 right:2
subtracter:: left:12 right:3
multiplier:: left:8 right:3
divider:: left:70 right:10
multiplier:: left:2 right:7
subtracter:: left:24 right:14
adder:: left:9 right:10
adder:: left:-1 right:19
```

- Flyweight

When a class requires multiple instantiation and all have common properties only a few of them differs then it's wise to instantiate only one object. Meanwhile if most of the properties are not common, among them would be provided externally; in this kind of situation flyweight pattern is used. Objects common property is maintained only once in memory whereas, properties different for each instantiation are passed from outside. The common intrinsic property save a memory, while extrinsic property is passed only when required. For example, in a housing colony map there are many houses in the map spread across various sectors and plots, but houses would be in certain types only. Type A, type B, type C, and type D(say). Each type has similar look and feel so graphic data structure needs to be created only once for each type where same graphic is shown at different places in the map and only vary in space coordinates. House types are intrinsic characteristics while house location is extrinsic characteristics. An army officer does not know each solder individually but he knows about different types of soldier groups (battalions) with different attires. An Army officer thinks about deploying soldier groups to various places in war.



Code

```
#include <iostream>
#include <map>
using namespace std;


class housetype {
string type;
public:
housetype(){}
housetype(const string& typep):type(typep){}
const string& gethousetype(){
return type;
}
};

class locationcontext {
int _sector,_plot;
public:
locationcontext(int sectorp,int plotp):_sector(sectorp),_plot(plotp){}
```

```
int sector() const {return _sector;}
int plot() const {return _plot;}
};

class house:public housetype {
public:
house(){};
house(const string& type):housetype(type){}
void build(const locationcontext& lcp){
cout<<"house with type : "<<gethousetype()<<" constructed at sector number : "<<lcp.sector()<<", plot number : "<<lcp.plot()<<endl;
}
};

class housefactory{
map<string,house> typemap;
public:
house& gethouse(const string& housetype){
if (!typemap.count(housetype))
typemap[housetype]=house(housetype);
return  typemap[housetype];
}
};

int main(void){
housefactory housefactoryi;
housefactoryi.gethouse("A").build(locationcontext(10, 1));
housefactoryi.gethouse("B").build(locationcontext(10, 2));
housefactoryi.gethouse("A").build(locationcontext(10, 3));
housefactoryi.gethouse("A").build(locationcontext(10, 4));
housefactoryi.gethouse("B").build(locationcontext(10, 5));
housefactoryi.gethouse("B").build(locationcontext(10, 7));
housefactoryi.gethouse("C").build(locationcontext(11, 1));
housefactoryi.gethouse("C").build(locationcontext(11, 2));
housefactoryi.gethouse("C").build(locationcontext(11, 4));
housefactoryi.gethouse("D").build(locationcontext(11, 5));
housefactoryi.gethouse("D").build(locationcontext(11, 7));
return 0;
}
```

Output

```
house with type : A constructed at sector number : 10, plot number : 1
house with type : B constructed at sector number : 10, plot number : 2
house with type : A constructed at sector number : 10, plot number : 3
house with type : A constructed at sector number : 10, plot number : 4
house with type : B constructed at sector number : 10, plot number : 5
house with type : B constructed at sector number : 10, plot number : 7
house with type : C constructed at sector number : 11, plot number : 1
house with type : C constructed at sector number : 11, plot number : 2
house with type : C constructed at sector number : 11, plot number : 4
house with type : D constructed at sector number : 11, plot number : 5
house with type : D constructed at sector number : 11, plot number : 7
```

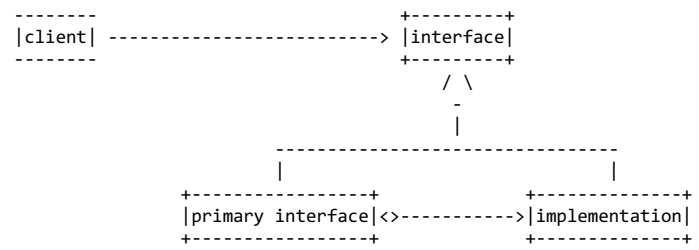
Day 2 Morning

5. GOF Structural Design Pattern

- Composition patterns
  - Adapter pattern
  - Bridge pattern
  - Facade pattern
  - Flyweight pattern
  - Proxy pattern
- Inheritance pattern
  - Proxy pattern
  - Adapter pattern
- Compostion + Inheritance patterns
  - Composite pattern
  - Decorator pattern

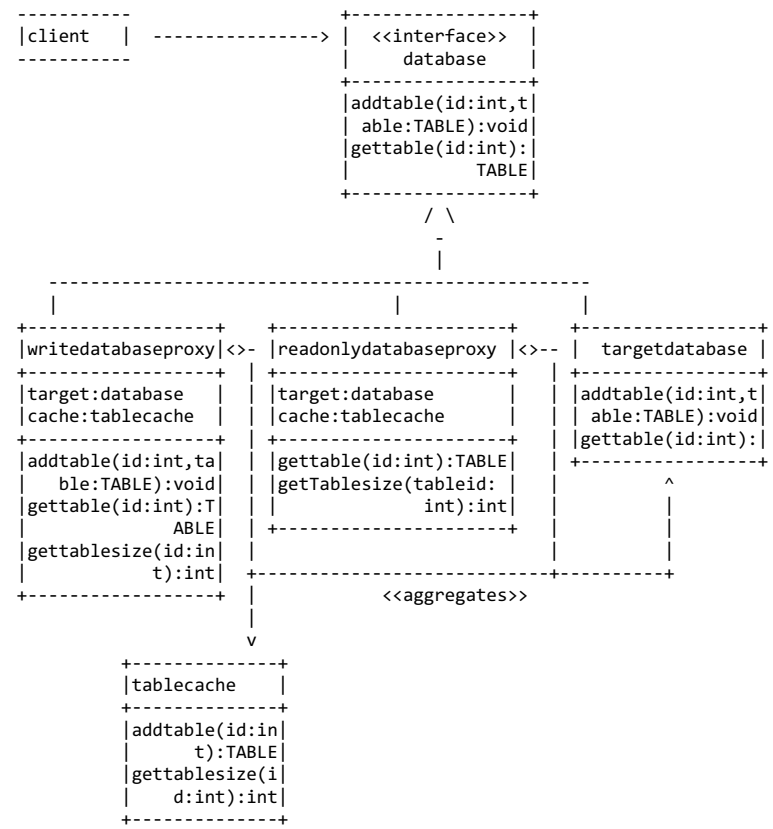
Inheritance

In this structural pattern, classes grow their attributes through inheritance rather than composition. This makes attributes static at run time. This pattern is used when either primary interface and delegatee both have same interface methods or primary interface implementation is derived from delegatee implementation, in this case delegatee methods would not extend.



- Proxy

A Class that acts as an interface to other class or classes is called proxy. There can be a scenario when target class is expensive to duplicate and a virtual class needs to be instantiated to many places but in return passes the call to target class for service. For example, a database program server requests based on query. It captures database in memory in order to provide the service. Loading database is a costly operation and may not be possible at low end machines And so instantiating the database class would load the database. Since memory consumption is high, this class can typically be instantiated at high end servers only. Programs in other machines that need database service cannot instantiate database class. Virtual database class is crucial as it provides exactly same interface as real database class but internally keeps cached information and serves the client internally and when required it gives real database program instance for the service. For the client programs they only instantiate real database programs (classes) and use same methods. Every country has an immigration department and some of them provide permission to stay in their country when you visit the country and they do not place their proxies in other countries.




```
#include <iostream>
#include <map>
#include <vector>
using namespace std;

class targetdatabase;
class tablecache;

class database {
public:
static targetdatabase* _tdb;
static tablecache* _tch;
targetdatabase* gettargetdatabase();
tablecache* gettablecache();
virtual void addtable(const string&, const vector<int>&)=0;
virtual vector<vector<int>>& gettable(const string&)=0;
virtual int gettablesize(const string&)=0;
};


targetdatabase* database::_tdb=NULL;
tablecache* database::_tch=NULL;

class targetdatabase:public database {
map<string,vector<vector<int>>> tablemap;
public:
void addtable(const string& tablename, const vector<int>& datap){
if (!tablemap.count(tablename))
 tablemap[tablename]=vector<vector<int>>();
tablemap[tablename].push_back(datap);
```

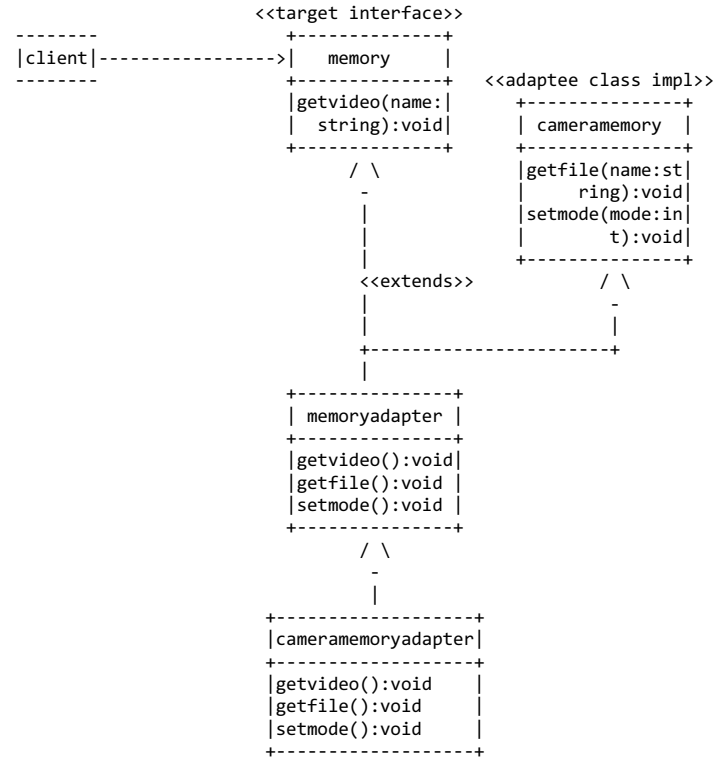
```
}
vector<vector<int>>& gettable(const string& tablename){
return tablemap[tablename];
}
int gettablesize(const string& tablename){
return tablemap[tablename].size();
}
};


class tablecache {
map<string,int> tablemap;
int size;
public:
void addtable(const string& namep, int sizep){
if (!tablemap.count(namep))
tablemap[namep]=sizep;
else
tablemap[namep]+=sizep;
}
int getsize(const string& tablenamep){
return tablemap[tablenamep];
}
};

targetdatabase* database::gettargetdatabase(){
if (!_tdb)
_tdb=new targetdatabase;
return _tdb;
}
tablecache* database::gettablecache(){
if (!_tch)
_tch=new tablecache;
return _tch;
}

class writedatabaseproxy:public database{
public:
void addtable(const string& name, const vector<int>& datap){
gettargetdatabase()->addtable(name,datap);
gettablecache()->addtable(name,1);
}
vector<vector<int>>& gettable(const string& namep){
return gettargetdatabase()->gettable(namep);
}
int gettablesize(const string& tablename){
gettablecache()->getsize(tablename);
}
};

class readdatabaseproxy:public database{
public:
void addtable(const string& name, const vector<int>& datap){
}
vector<vector<int>>& gettable(const string& namep){
return gettargetdatabase()->gettable(namep);
}

int gettablesize(const string& tablename){
gettablecache()->getsize(tablename);
}
};

int main(void){
writedatabaseproxy wdp;
readdatabaseproxy rdp;
wdp.addtable("one",vector<int>{1, 2, 3});
wdp.addtable("one",vector<int>{4, 5, 6});
wdp.addtable("one",vector<int>{7, 8, 9});
wdp.addtable("two",vector<int>{11, 12, 13});
wdp.addtable("two",vector<int>{14, 15, 16});
wdp.addtable("two",vector<int>{17, 18, 19});
wdp.addtable("two",vector<int>{20, 21, 22});
cout<<"table size for tableid: one :"<<rdp.gettablesize("one")<<endl;
cout<<"table size for tableid: two :"<<rdp.gettablesize("two")<<endl;
cout<<"table data for tableid: two :"<<endl;
for(int i=0;i<rdp.gettable("two").size();i++){//<<rdp.gettable("two")<<endl;
 for(int j=0;j<rdp.gettable("two")[i].size();j++)
  cout<<rdp.gettable("two")[i][j]<<" ";
 cout<<endl;
}
return 0;
}
```

Output

```
table size for tableid: one :3
table size for tableid: two :4
table data for tableid: two :
11 12 13
14 15 16
17 18 19
20 21 22
```

- Adapter class

Adapter class type serves the same function as the adapter object type. adapter (and its subclasses) in addition to target it's also derived from the adaptee. This makes delegation easy whereas sub classing adaptee is not possible.



Code

```
#include <iostream>
#include <string>
using namespace std;

class memory{
public:
virtual const string& fetchvideo(const string&)=0;
};

class cameramemory {
public:
int mode;
string videofile;
virtual void setmode(int)=0;
virtual const string& getfile(const string&)=0;
};

class nikoncameramemory:public cameramemory {
public:
void setmode(int modep) {
mode=modep;
if (modep==1)
 cout<<"nikoncameramemory::CHANGED TO READ MODE"<<endl;
else
 cout<<"nikoncameramemory::CHANGED TO WRITE MODE"<<endl;
}
const string& getfile(const string& name) {
if (!mode)
 videofile="ERROR in MODE, can not read in write mode, change the mode";
else{
 cout<<"nikoncameramem::serving file : MEMFILE1"<<endl;
 videofile="MEMFILE1";
}
return videofile;
}
};

class memoryadapter:public memory,public nikoncameramemory {
public:
const string& fetchvideo(const string& name){
setmode(1);
return getfile(name);
}
};

int main(){
memory *madapter=new memoryadapter;
cout<<madapter->fetchvideo("earth song")<<endl;
```

```
delete madapter;
return 0;
}
```

Output

```
nikoncameramemory::CHANGED TO READ MODE
nikoncameramem::serving file : MEMFILE1
MEMFILE1
```

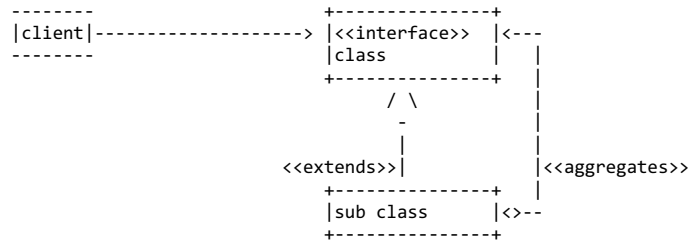
Day 2 Morning

5. GOF Structural Design Pattern

- Composition patterns
  - Adapter pattern
  - Bridge pattern
  - Facade pattern
  - Flyweight pattern
  - Proxy pattern
- Inheritance pattern
  - Proxy pattern
  - Adapter pattern
- Compostion + Inheritance patterns
  - Composite pattern
  - Decorator pattern

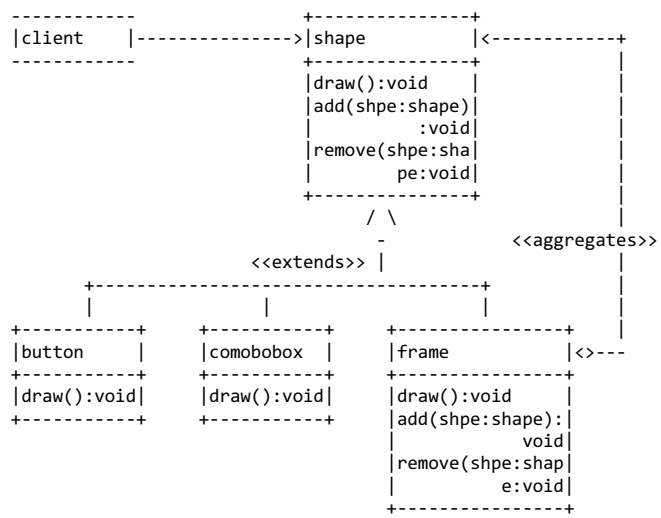
Composition + Inheritance

This kind of structural pattern consist of both inheritance and composition. sub class that extends the base class also compose the base class. In this sub class provide same interface methods as base class and in addition it composes other leaf subclasses that extend its attributes.



- Composite pattern

When a complex structure contains other structures where the other structures provide the same interface behaviour as the complex structure, this situation makes composite structural pattern; this is where a structure extends its attribute through similar interface (sub-classed from the same parent) leaf classes. When complex structure subclasses, it becomes recursive in nature; for example a graphic component (i.e frame) can compose other several graphics (button, checkbox, frame) and out of the composed, some composite graphic can contain similar objects (button, checkbox, frame). The frame is a composite graphic which also contains leaf (button, checkbox) graphic and composite graphic(frame). In an arithmetic expression operators are leaf elements whereas expression itself is composite element. Code example is provided in the below graphic example.



Code

```
#include <iostream>
#include <list>

using namespace std;
```

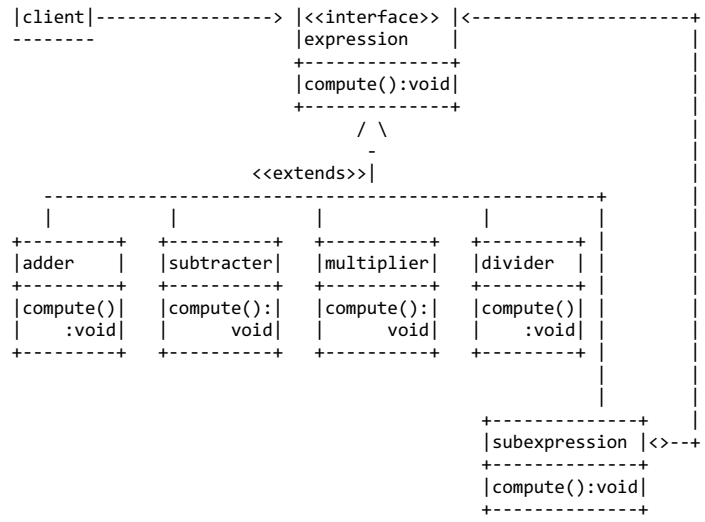
```
class shape{
public:
string id;
shape(const string& idp):id(idp){}
virtual void draw()=0;
virtual void add(shape*){};
virtual void remove(shape*){};
virtual ~shape(){};
};

class frame:public shape {
list<shape*> child;
public:
frame(const string& idp):shape(idp){}
void draw() {
list<shape*>::iterator it;
cout<<id<<"::draw()"<<endl;
for(it=child.begin();it!=child.end();it++)
 (*it)->draw();
}
void add(shape* framep){
child.push_front(framep);
}
void remove(shape* framep){
child.remove(framep);
}
virtual ~frame(){
cout<<"~"<<id<<endl;
list<shape*>::iterator it;
for(it=child.begin();it!=child.end();it++)
 delete (*it);
}
};

class button:public shape {
string text;
public:
button(const string& idp,const string& textp):shape(idp),text(textp){}
void draw(){
cout<<" "<<id<<"::draw() text: "<<text<<endl;
}
~button(){cout<<" ~"<<id<<endl;}
};

class combobox:public shape{
list<string> valuelist;
public:
combobox(const string& idp, const list<string>& valuelistp):shape(idp),valuelist(valuelistp){}
void draw(){
list<string>::iterator it;
cout<<" "<<id<<"::draw() elements: ";
for(it=valuelist.begin();it!=valuelist.end();it++)
cout<<(*it)<<" ";
cout<<endl;
}
~combobox(){cout<<" ~"<<id<<endl;}
};

int main(void){
frame* frametop=new frame("frametop");
frame* frameleft=new frame("frameleft");
frame* frameright=new frame("frameright");
frameleft->add(new combobox("comboboxleft",list<string>{"one","two","three"}));
frameleft->add(new button("buttonleft","OK"));
frameright->add(new combobox("comboright",list<string>{"animal","bird","reptile"}));
frameright->add(new button("buttonright","OK"));
frametop->add(frameleft);
frametop->add(frameright);
frametop->draw();
delete frametop;
return 0;
}
```

Output

```
frametop::draw()
frameright::draw()
 buttonright::draw() text: OK
 comboright::draw() elements: animal bird reptile
frameleft::draw()
 buttonleft::draw() text: OK
 comboboxleft::draw() elements: one two three
~frametop
~frameright
 ~buttonright
 ~comboright
~frameleft
 ~buttonleft
 ~comboboxleft
```

Example 2 (arithmetic expression):

```
 --------                    +--------------+
```



Code

```
#include <iostream>
#include <list>
#include <regex>
using namespace std;

class expression {
public:
expression(){}
virtual ~expression(){}
virtual string compute(string)=0;
};

class subexpression :public expression{
list<expression*> expressionlist;
public:
subexpression(const list<expression*>& expressionlistp):expressionlist(expressionlistp){}
string compute(string mep){
cout<<"subexpression : "<<mep<<endl;
list<expression*>::iterator it;
for (it=expressionlist.begin();it!=expressionlist.end();it++) mep=(*it)->compute(mep);
return mep;
}
~subexpression(){
list<expression*>::iterator it;
for (it=expressionlist.begin();it!=expressionlist.end();it++) delete (*it);
}
};

class adder:public expression {
public:
virtual string compute(string mep) {
smatch sm;
cout<<"adder : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
return mep;
}
};

class substracter:public expression {
public:
virtual string compute(string mep) {
smatch sm;
cout<<"substracter : "<<mep<<endl;
while(regex_search(mep,sm,regex(R"((.*?)(\d+)\-(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
return mep;
}
};

class multiplier:public expression {
public:
virtual string compute(string mep) {
smatch sm;
cout<<"multiplier : "<<mep<<endl;
while(regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
return mep;
}
};

class divider:public expression {
public:
virtual string compute(string mep) {
smatch sm;
cout<<"divider : "<<mep<<endl;
```

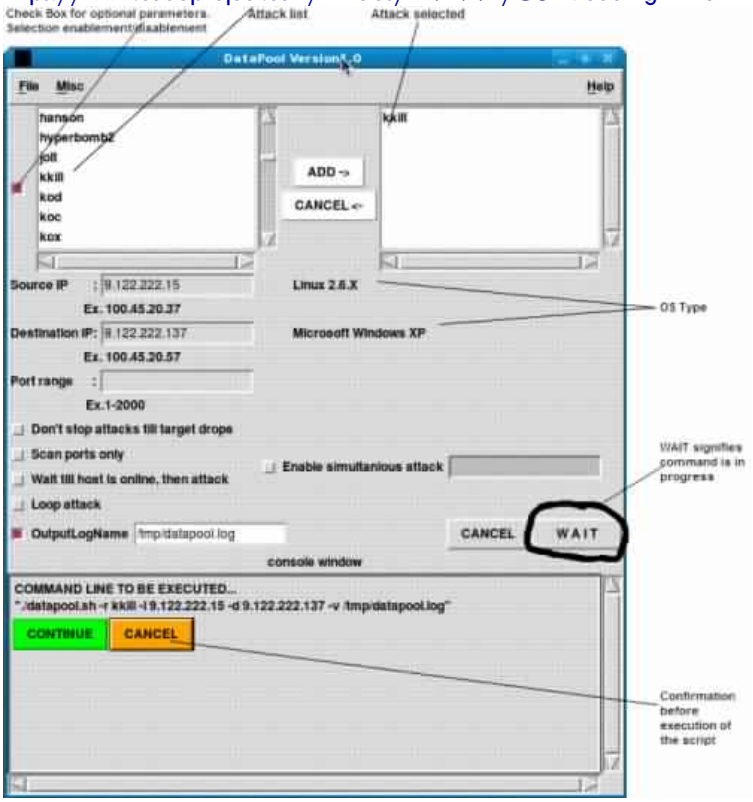
```
while(regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
return mep;
}
};

int main(void){
string result;
expression *exp=new subexpression(list<expression*>{new divider,new subexpression(list<expression*>{new multiplier, new
subexpression(list<expression*>{new substracter, new adder})})});
result=exp->compute("1+3/3*2-2+6/2/3-2");
cout<<"result : "<<result<<endl;
result=exp->compute("1-2+4*3-6/2+8*3-2*70/10");
cout<<"result : "<<result<<endl;
delete exp;
return 0;
}
```

Output

```
subexpression : 1+3/3*2-2+6/2/3-2
divider : 1+3/3*2-2+6/2/3-2
subexpression : 1+1*2-2+1-2
multiplier : 1+1*2-2+1-2
subexpression : 1+2-2+1-2
substracter : 1+2-2+1-2
adder : 1+0+-1
result : 0
subexpression : 1-2+4*3-6/2+8*3-2*70/10
divider : 1-2+4*3-6/2+8*3-2*70/10
subexpression : 1-2+4*3-3+8*3-2*7
multiplier : 1-2+4*3-3+8*3-2*7
subexpression : 1-2+12-3+24-14
substracter : 1-2+12-3+24-14
adder : -1+9+10
result : 18
```

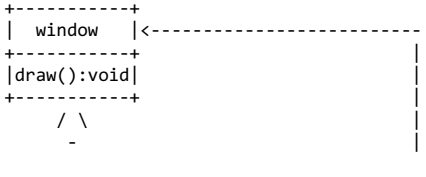
Article

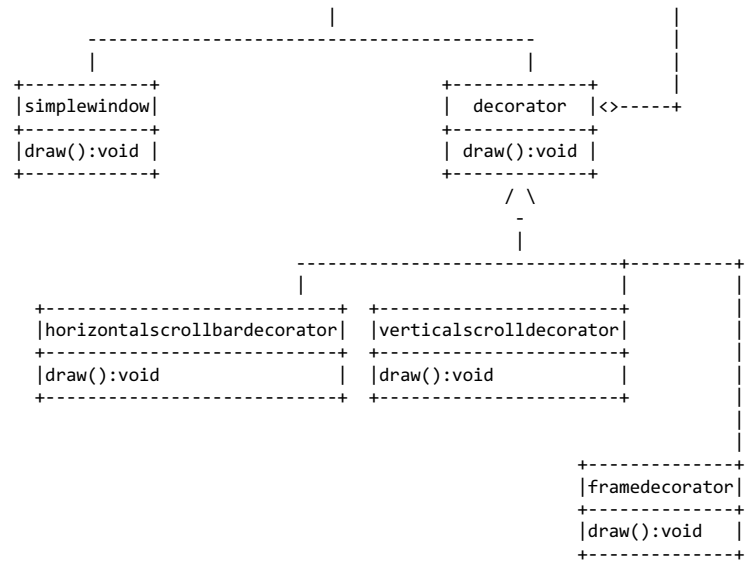
https://www.codeproject.com/Articles/1271791/GUI-Modeling-in-Perl-Tk-Using-Composite-Design-Pat



- Decorator

When extending attributes are required at run time, a similar pattern to composite can work. It's class in which attributes have to be extended and instead of composite, the decorator classes extends the attributes of the class. A decorator is further sub-classed in order to have more attributes in it. For example for a leaf class window, a decorator class can be a frame window drawing around it which is then sub-classed to vertical scrollbar that provides vertical scrollbar for the frame. A house knows how to show itself but when it gets lawns and swimming pool around it as decorators, it gets elegant.





Code

```
#include <iostream>
using namespace std;

class window {
public:
virtual void draw()=0;
virtual ~window(){};
};

class simplewindow:public window {
public:
void draw(){
cout<<"simplewindow";
}
~simplewindow(){cout<<"~simplewindow ";}
};

class decorator:public window {
window *decoratee;
public:
decorator(window* decorateep):decoratee(decorateep){}
void draw(){
decoratee->draw();
}
~decorator(){delete decoratee;}
};

class framedecorator:public decorator {
public:
framedecorator(window* decorateep):decorator(decorateep){}
void draw(){
decorator::draw();
cout<<" with frame";
}
~framedecorator(){cout<<"~framedecorator ";}
};

class horizontalscrollbardecorator:public decorator {
public:
horizontalscrollbardecorator(window* decorateep):decorator(decorateep){}
void draw(){
decorator::draw();
cout<<" with horizontalscrollbar";
}
~horizontalscrollbardecorator(){cout<<"~horizontalscrollbardecorator ";}
};

class verticalscrollbardecorator:public decorator {
public:
verticalscrollbardecorator(window* decorateep):decorator(decorateep){}
void draw(){
decorator::draw();
cout<<" with verticalscrollbar";
}
~verticalscrollbardecorator(){cout<<"~verticalscrollbardecorator ";}
};

int main(void){
window *win=new horizontalscrollbardecorator(new verticalscrollbardecorator(new framedecorator(new simplewindow)));
win->draw();
cout<<endl;
delete win;
cout<<endl;
```

```
return 0;
}
```

Output

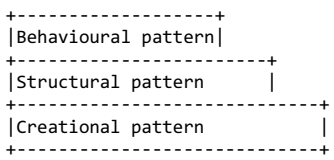
```
simplewindow with frame with verticalscrollbar with horizontalscrollbar
~horizontalscrollbardecorator ~verticalscrollbardecorator ~framedecorator ~simplewindow
```

Day 3 Morning

6. GOF Behavioural Design Pattern - 1

- Behavioral design patterns
  - Recursive method calls
    - Chain of responsibility pattern
    - Interpreter pattern
  - Non recursive calls
    - Callback method calls
    - Command pattern
    - State pattern
    - Observer pattern
    - Visitor pattern

Behavioral design pattern include patterns that focus on operations (activity) of a class. Every class/ component is known for the operations it performs and the behavior it attributes but not how it is structured or created. Unlike structural pattern which has base on creational pattern, behavioral pattern has base on structural pattern.



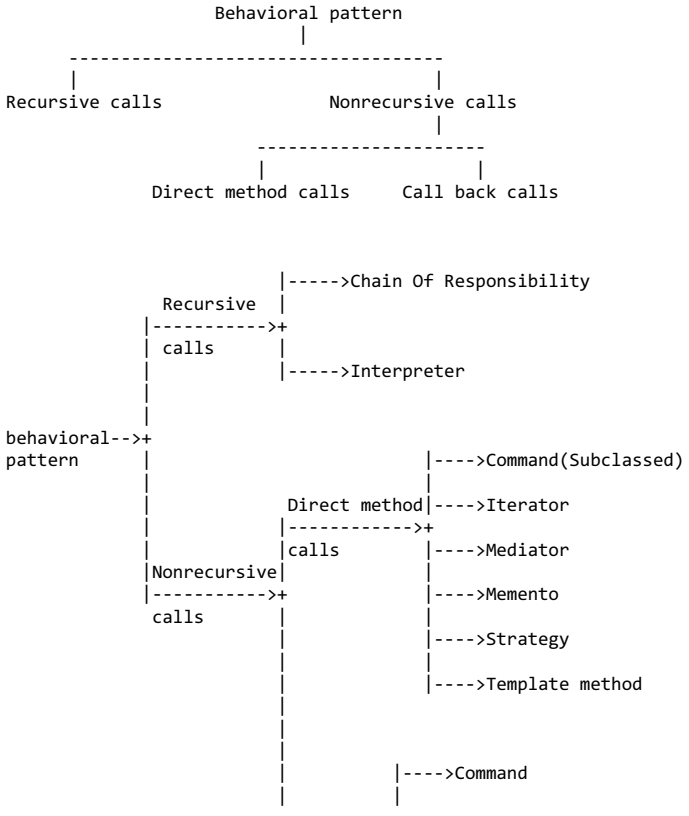
As this pattern focuses on operations carried out by the pattern, it mostly deals with how a class method is called or how a set of class methods call each other in order to show a different behavior.

For example if we register a class method in a class and when a method of the class make callback to the function that's called command pattern.

Another type can be when a class method(function) calls recursively the same function of its own type of class object, the recursive way of calling the function and chain of responsibility is a pattern which matches to this. Behavioral patterns can be distributed in two main categories.

a) Recursive method calls.
b) Non recursive method call.

Non-recursive calls can be further distributed to:
- o Direct method calls.
- o Call back methods calls.



© www.minhinc.com

```
|Callback  |---->State
|--------->+
 calls     |---->Observer
           |
           |---->Visitor
```

Day 3 Morning

6. GOF Behavioural Design Pattern - 1

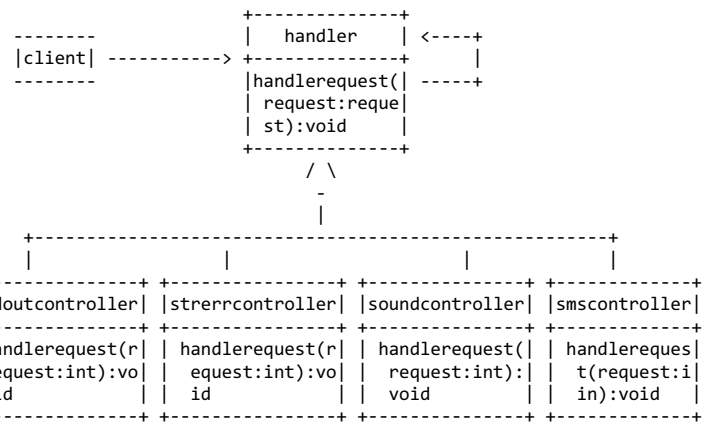
- Behavioral design patterns
- Recursive method calls
  - Chain of responsibility pattern
  - Interpreter pattern
- Non recursive calls
  - Callback method calls
  - Command pattern
  - State pattern
  - Observer pattern
  - Visitor pattern

Recursive patterns
As per this pattern type, a method calls itself or calls the same method in other Sub-classes.

- Chain of responsibility
Responsibility to handle a request is passed to objects in a chain until the request finds an object that can handle it. In addition the Object has to stop the message flow or pass it to the next object in the chain. This kind of pattern is called chain of responsibility. It avoids attaching the sender of a request to a particular receiver and gives a generic way to get the request handled. The sender needs only to worry about the first element of the chain in which the request will pass through.. For example in fault handling section, when a fault occurs, it can either be displayed as a standard message, an error message, produce sound alarm, send a message to higher officials for immediate actions or it can be any combination of the above actions.

Rather than taking actions from every separate individual separate entities, there can be separate entity for each of the actions and in the event of fault occurrence it's passed in a chain to the first entity to take actions. Every entity in the chain checks the severity of the fault and decides on the handling of the fault. Another example, an arithmetic expression can be solved through this pattern, below is a code example for this explanation. In another scenario a number can be shuffled.

A Code example follows the arithmetic expression example. In a football match a goalkeeper passes the ball to the next player without actually knowing who will pass the ball in the other side of the goalpost.




```
#include <iostream>
using namespace std;

class handler {
handler *next;
public:
handler(handler *nextp):next(nextp){}
virtual void handle(int valuep){if(next)next->handle(valuep);}
virtual ~handler(){if(next)delete next;};
};

class stdoutcontroller:public handler{
public:
```

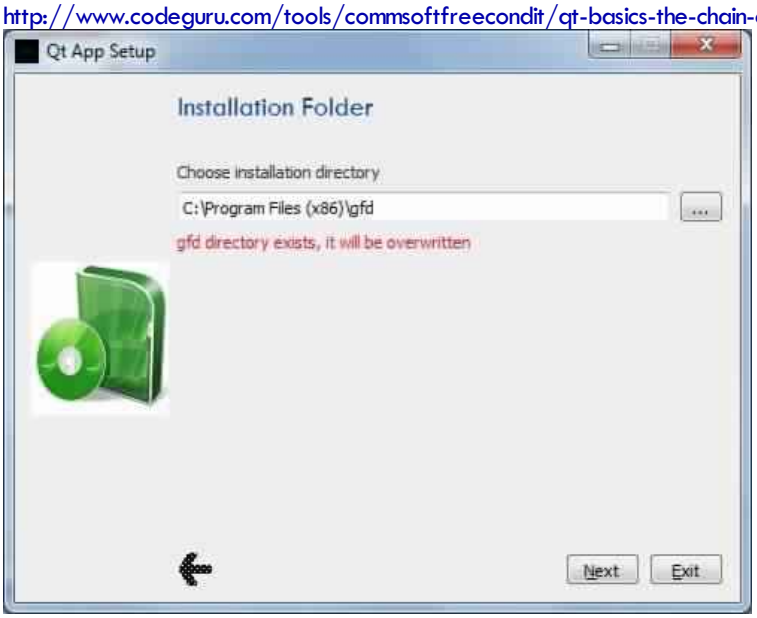
```
stdoutcontroller(handler *nextp):handler(nextp){}
virtual void handle(int valuep) {
cout<<"stdoutcontroller::handlerequest"<<endl;
handler::handle(valuep);
}
~stdoutcontroller(){cout<<"~stdoutcontroller"<<endl;}
};

class stderrorcontroller:public handler{
public:
stderrorcontroller(handler *nextp):handler(nextp){}
virtual void handle(int valuep) {
cout<<"stderrorcontroller::handlerequest"<<endl;
handler::handle(valuep);
}
~stderrorcontroller(){cout<<"~stderrorcontroller"<<endl;}
};

class soundcontroller:public handler{
public:
soundcontroller(handler *nextp):handler(nextp){}
virtual void handle(int valuep) {
cout<<"soundcontroller::handlerequest"<<endl;
handler::handle(valuep);
}
~soundcontroller(){cout<<"~soundcontroller"<<endl;}
};

class smscontroller:public handler{
public:
smscontroller(handler *nextp):handler(nextp){}
virtual void handle(int valuep) {
cout<<"smscontroller::handlerequest"<<endl;
handler::handle(valuep);
}
~smscontroller(){cout<<"~smscontroller"<<endl;}
};


int main(void){
handler* chain=new stdoutcontroller(new stderrorcontroller(new soundcontroller (new smscontroller(NULL))));
chain->handle(10);
delete chain;
return 0;
}
```

Output

```
stdoutcontroller::handlerequest
stderrorcontroller::handlerequest
soundcontroller::handlerequest
smscontroller::handlerequest
~stdoutcontroller
~stderrorcontroller
~soundcontroller
~smscontroller
```

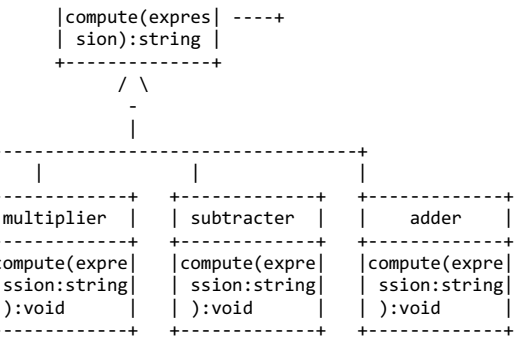
Article

http://www.codeguru.com/tools/commsoftfreecondit/qt-basics-the-chain-of-responsibility-pattern.html



Second example(Arithmetic calculation):

```
--------                +--------------+
|client| --------------> |   handler    | <---+
--------                +--------------+     |
```



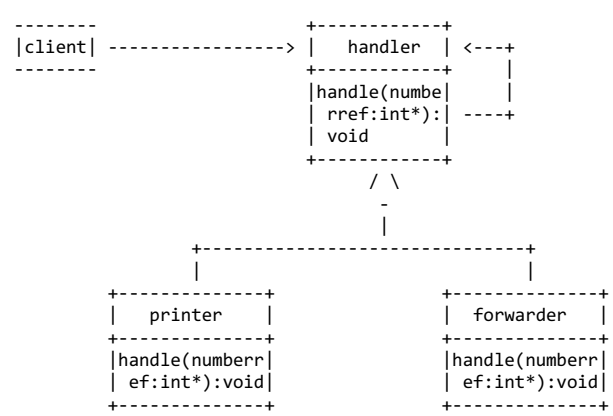
Code

```
#include <iostream>
#include <regex>

using namespace std;
class handler{
handler *next;
public:
handler(handler *nextp):next(nextp){}
virtual string handle(string mep){if (next) return next->handle(mep); else return mep;}
~handler(){if (next) delete next;}
};

class adder:public handler {
public:
using handler::handler;
string handle(string mep) {
smatch sm;
cout<<"adder : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
return handler::handle(mep);
}
};

class substracter:public handler {
public:
using handler::handler;
string handle(string mep) {
smatch sm;
cout<<"adder : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\-(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
return handler::handle(mep);
}
};

class multiplier:public handler {
public:
using handler::handler;
string handle(string mep) {
smatch sm;
cout<<"adder : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
return handler::handle(mep);
}
};

class divider:public handler {
public:
using handler::handler;
string handle(string mep) {
smatch sm;
cout<<"adder : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
return handler::handle(mep);
}
};


int main(void){
string result;
handler *hndlr=new divider(new multiplier(new substracter(new adder(NULL))));
result=hndlr->handle("1+3/3*2-2+6/2/3-2");
cout<<"result : "<<result<<endl;
result=hndlr->handle("1-2+4*3-6/2+8*3-2*70/10");
cout<<"result : "<<result<<endl;
delete hndlr;
return 0;
}
```

Output

```
adder : 1+3/3*2-2+6/2/3-2
adder : 1+1*2-2+1-2
adder : 1+2-2+1-2
adder : 1+0+-1
result : 0
adder : 1-2+4*3-6/2+8*3-2*70/10
adder : 1-2+4*3-3+8*3-2*7
adder : 1-2+12-3+24-14
adder : -1+9+10
result : 18
```

Third example(Reshuffling number):



Code

```
#include <iostream>
#include <regex>

using namespace std;
class handler {
public:
handler *next;
virtual void handle(const string)=0;
~handler(){if(next) delete next;}
};

class printer:public handler {
public:
void handle(const string exp){
cout<<exp<<endl;
};
};

class fowarder:public handler {
int _pos;
public:
fowarder(int posp,int sizep):_pos(posp){
if(_pos>1)
next=new fowarder(_pos-1,sizep);
else
next=new printer;
}
void handle(const string exp) {
int count=0;
next->handle(exp);
for(count=0;count<_pos;count++) next->handle(regex_replace(exp,regex(string("(.{")+to_string(exp.size()-_pos-1)+string("})(.)(.
{")+to_string(count)+string("})(.)(.*)")),"$1$4$3$2$5"));
}
};


int main(void){
string str="1234ab";
handler *sh=new fowarder(str.size()-1,str.size());
sh->handle(str);
delete sh;
return 0;
}
```

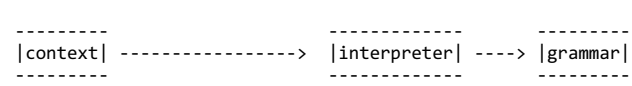
Output

```
1234ab
1234ba
123a4b
123ab4
123ba4
123b4a
1243ab
1243ba
124a3b
124ab3
124ba3
124b3a
12a43b
12a4b3
12a34b
12a3b4
12ab34
12ab43
12b4a3
.
.
132b4a
.
```

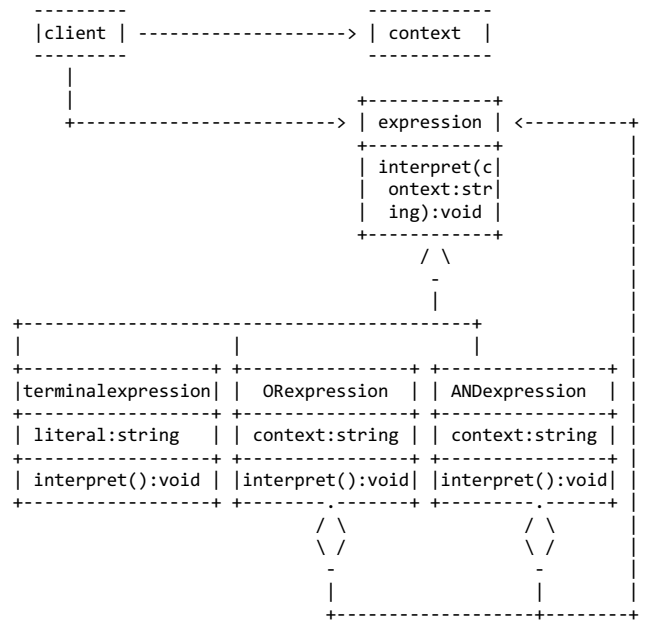
.
.
b1432a
b14a32
b14a23
b142a3
b1423a
b1a432
b1a423
b1a342
b1a324
b1a234
b1a243
b124a3
b1243a
b12a43
b12a34
b123a4
b1234a

-Interpreter

A grammar interpreter pattern is used in a language where representation of grammar is required along with
an interpreter in-order to decode sentences in a language. For example in a language there are grammar/rules
for rahul AND (abdul OR (ravi OR john) AND sally.
The grammar/rule says rahul and sally and any of abdul or ravi or john. In the case of a context this grammar
will be used by the interpreter in order to evaluate the context. when we hear a new word, we interpret it
through a dictionary.



An interpreter makes reference to grammar in order to evaluate context.
Every religion has a grammar book.



Code

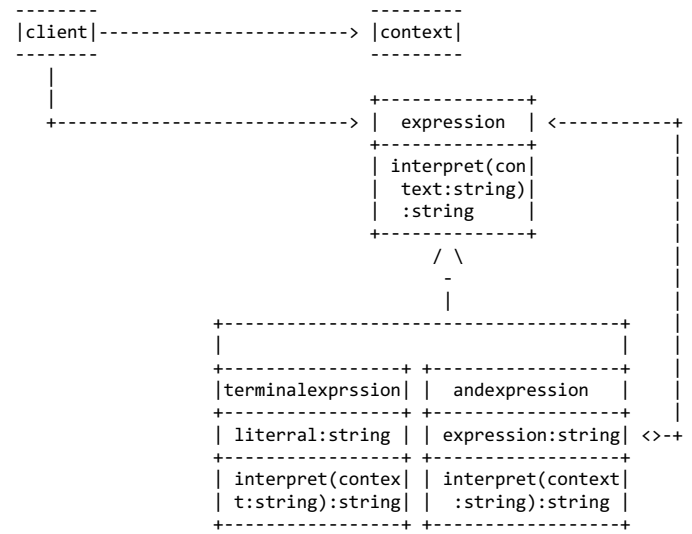
```
#include <iostream>
#include <regex>

using namespace std;
class expression{
public:
expression *exp1,*exp2;
expression():exp1(NULL),exp2(NULL){}
expression(expression* exp1p,expression* exp2p):exp1(exp1p),exp2(exp2p){}
virtual bool interpret(const string&)=0;
virtual ~expression(){
if (exp1 && exp2){
delete exp1;
delete exp2;
}
}
};

class terminalexpression:public expression{
string literal;
public:
terminalexpression(const string& literalp):literal(literalp){}
```

```
bool interpret(const string& contextp){
regex b(string(".*\\b")+literal+string("\\b.*"));
if (regex_match(contextp,b))
 return true;
else
 return false;
}
};

class ORexpression:public expression {
public:
ORexpression(expression* exp1p,expression* exp2p):expression(exp1p,exp2p){}
virtual bool interpret(const string& contextp){
return exp1->interpret(contextp) || exp2->interpret(contextp);
}
};

class ANDexpression:public expression {
public:
ANDexpression(expression* exp1p,expression* exp2p):expression(exp1p,exp2p){}
bool interpret(const string& contextp){
return exp1->interpret(contextp) && exp2->interpret(contextp);
}
};

int main(void){
cout<<"Grammar expression : \"RAHUL AND (ABDUL OR ((RAVI OR JOHN) AND UDONG)\""<<endl;
expression *expressioni=new ANDexpression(new terminalexpression("RAHUL"),new ORexpression(new terminalexpression("ABDUL"),new
ANDexpression(new ORexpression(new terminalexpression("RAVI"),new terminalexpression("JOHN")),new terminalexpression("UDONG"))));
cout<<"interpreting \"JOHN AND UDONG AND RAHUL\" : "<<expressioni->interpret("JOHN AND UDONG AND RAHUL")<<endl;
cout<<"interpreting \"RAHUL AND RAVI\" : "<<expressioni->interpret("RAHUL AND RAVI")<<endl;
delete expressioni;
return 0;
}
```

Output

```
Grammar expression : "RAHUL AND (ABDUL OR ((RAVI OR JOHN) AND UDONG)"
interpreting "JOHN AND UDONG AND RAHUL" : 1
interpreting "RAHUL AND RAVI" : 0
```

Arithmetic expression computation:



Code

```
#include <iostream>
#include <regex>
#include <functional>

using namespace std;
class expression {
public:
virtual string interpret(string)=0;
};

class terminalexpression : public expression {
std::function<int(int,int)> op;
char oprtr;
public:
terminalexpression(const std::function<int(int,int)>& opp,char oprtrp):op(opp),oprtr(oprtrp){}
string interpret(string mep){
smatch sm;
cout<<oprtr<<" : "<<mep<<endl;
while (regex_search(mep,sm,regex(string("(.*?)(-?\\d+)\\")+oprtr+string("(-?\\d+)(.*)"))))
mep=sm.str(1)+to_string(op(stoi(sm.str(2)),stoi(sm.str(3))))+sm.str(4);
```

```
return mep;
}
};

class andexpression:public expression {
expression *exp1,*exp2;
public:
andexpression(expression* exp1p,expression* exp2p):exp1(exp1p),exp2(exp2p){}
string interpret(string mep){
return exp2->interpret(exp1->interpret(mep));
}
};

int main(void){
expression *exp = new andexpression(new andexpression(new terminalexpression(std::divides<int>(),'/'),new
terminalexpression(std::multiplies<int>(),'*')),new andexpression(new terminalexpression(std::minus<int>(),'-'),new
terminalexpression(std::plus<int>(),'+')));
string result;
result=exp->interpret("1+3/3*2-2+6/2/3-2");
cout<<"result : "<<result<<endl;
result=exp->interpret("1-2+4*3-6/2+8*3-2*70/10");
cout<<"result : "<<result<<endl;
return 0;
}
```

Output

```
/ : 1+3/3*2-2+6/2/3-2
* : 1+1*2-2+1-2
- : 1+2-2+1-2
+ : 1+0+-1
result : 0
/ : 1-2+4*3-6/2+8*3-2*70/10
* : 1-2+4*3-3+8*3-2*7
- : 1-2+12-3+24-14
+ : -1+9+10
result : 18
```

Day 3 Morning

6. GOF Behavioural Design Pattern - 1

- Behavioral design patterns
- Recursive method calls
  - Chain of responsibility pattern
  - Interpreter pattern
- Non recursive calls
  - Callback method calls
    - Command pattern
    - State pattern
    - Observer pattern
    - Visitor pattern

Non recursive patterns
In this kind of patterns, a method calls the other in linear (direct call) fashion or because they are already registered(call back).

Callback patterns
This is when the name of the callee is already registered with the caller and some event triggers the callers to callback the callee.

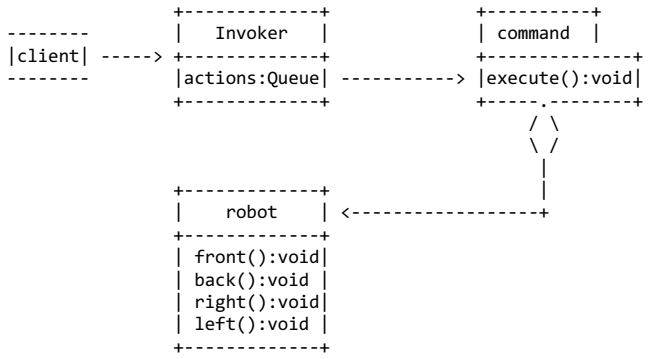
- Command

Command pattern is used when different methods need to be called in a generic fashion. In command, methods get class status and executing class generic method actually calls different class methods for which the command object represents. There can be two types of pattern:

1. Separate command class, registers the methods of a subject class and executes its (subject class) methods once execute method of command is clicked,

2. Command class is sub-classed to serve different behaviors and the program needs to mix code with abstract command class. Execute method and the actual execute are called in the context of sub-classed behaviors. Lower level staffs in every department are command executers for commands received from various higher officials.

First approach: Class methods register their methods with the command class and when execute method of command class is executed a callback to already registered method is send. For example a robot makes many movements; front, back, left, and right. A function executing these movements need these operations stored in a list so as to execute the list one by one as the movement takes on. The function does not know about the robot and its particular movements. This can happen when robot actions get registered in a new class called command class and when client executes its execute method, through callback robot action.



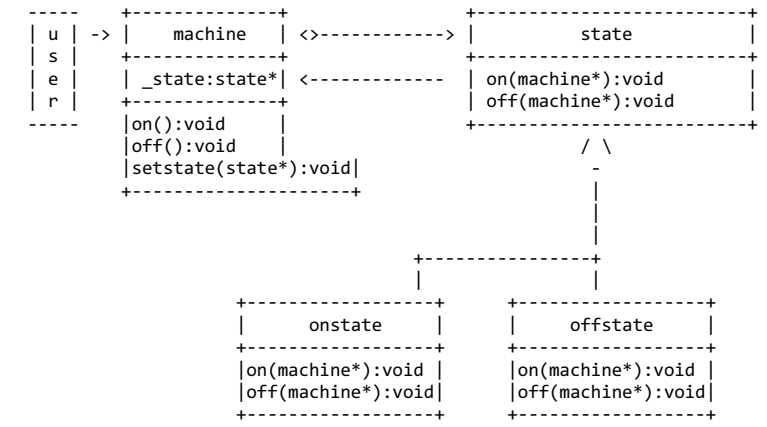

```
#include <iostream>
#include <list>

using namespace std;
class robot;
class command {
robot* rbt;
void (robot::*method)();
public:
command(robot* rbtp, void (robot::*methodp)()){
rbt=rbtp;
method=methodp;
}
virtual void execute(){
(rbt->*method)();
}
};

class robot {
public:
void left(){
cout<<"robot move left"<<endl;
}
void right(){
cout<<"robot move right"<<endl;
}
void front(){
cout<<"robot move front"<<endl;
}
void back() {
cout<<"robot move back"<<endl;
}
};

class invoker{
public:
void action(const list<command*>& commandlistp) {
list<command*>::const_iterator it;
for (it=commandlistp.begin();it!=commandlistp.end();it++)
(*it)->execute();
}
};

int main(void){
invoker invokeri;
robot roboti;
invokeri.action(list<command*>{new command(&roboti,&robot::left),new command(&roboti,&robot::front),new
command(&roboti,&robot::right),new command(&roboti,&robot::back)});
return 0;
}
```

Output

```
robot move left
robot move front
robot move right
robot move back
```

- State

A class behavior may change when the state of data type changes, so class function does different operation depending on the state of the data type. Behavior code is bound to change when performing an operation based on state code, where state is separated from the main behavior and when required behavior passes logic to a separate state abstraction. This type of state abstraction is called state pattern. This makes behavior and logic separated, and logic is in state classes making behavior bound to changes and open for logic to be extended in shape of more state sub-classes possible to be added at run time. For example a lift can sustain 5 people. A person can only use the lift when there are less than 5 people in the lift. A Lift operates through opening and closing of lift door, and getting lift move up and down. Lift states include lift at rest, lift serving people up, lift serving people down, and lift full. In electrical switch board there is a plug point and a switch. When a user inputs pin in an electrical equipment in the plug, the board passes the information to the switch. The switch internally maintains various states (i.e. on and off), one state at a time would be effective and the active state receives the information. It is the state which decides the action of the plug and typically 'on' allows the connection; when 'off' it's in still state and cannot allow any action.



Code

```
#include <iostream>
using namespace std;

class machine;
class state{
public:
virtual void on(machine *m)=0;
virtual void off(machine *m)=0;
};

class machine{
state* _state;
public:
~machine(){delete _state;}
void setstate(state* statep){_state=statep;}
void on();
void off();
};

class onstate:public state{
public:
void on(machine *m){
cout<<"Already in ON state"<<endl;
}
void off(machine*);
};

class offstate:public state{
public:
void on(machine *m){
cout<<"OFF -> ON"<<endl;
m->setstate(new onstate);
delete this;
}
void off(machine *m){
cout<<"Already in OFF state"<<endl;
}
};

void onstate::off(machine *m){
cout<<"ON -> OFF"<<endl;
m->setstate(new offstate);
delete this;
}
void machine::on(){_state->on(this);}
void machine::off(){_state->off(this);}
```

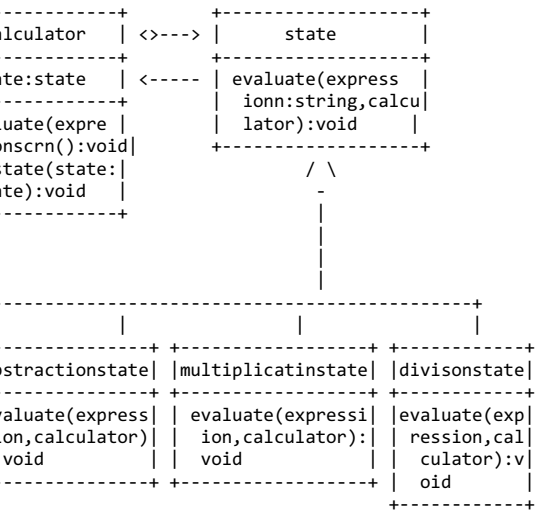
© www.minhinc.com

```
int main(void){
machine machinei;
machinei.setstate(new offstate);
machinei.on();
machinei.on();
machinei.off();
machinei.on();
return 0;
};
```

Output

```
OFF -> ON
Already in ON state
ON -> OFF
OFF -> ON
```

State pattern way of solving arithmetic equation:



Code

```
#include <iostream>
#include <regex>

using namespace std;
class calculator;
class state{
public:
virtual void evaluate(string&,calculator*)=0;
};
class calculator{
state *_state;
public:
void setstate(state *statep){_state=statep;}
string evaluate(string mep){ while(_state!=NULL) _state->evaluate(mep,this); return mep; }
};

class addstate:public state{
public:
void evaluate(string& mep,calculator *calcp){
smatch sm;
cout<<"addstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
calcp->setstate(NULL);
delete this;
}
};

class substractstate:public state{
public:
void evaluate(string& mep,calculator *calcp){
smatch sm;
cout<<"substractstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\-(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
calcp->setstate(new addstate);
delete this;
}
};

class multiplierstate:public state{
public:
void evaluate(string& mep,calculator *calcp){
smatch sm;
cout<<"multiplierstate : "<<mep<<endl;
```

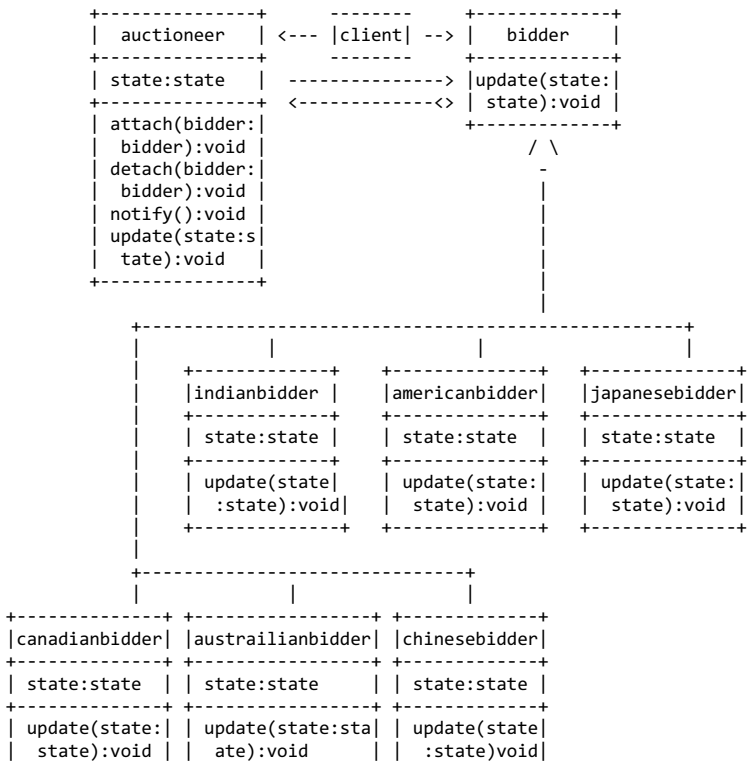
```
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
calcp->setstate(new substractstate);
delete this;
}
};

class dividestate:public state{
public:
void evaluate(string& mep,calculator *calcp){
smatch sm;
cout<<"dividetstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
calcp->setstate(new multiplierstate);
delete this;
}
};

int main(void){
calculator calc;
string result;
calc.setstate(new dividestate);
result=calc.evaluate("1+3/3*2-2+6/2/3-2");
cout<<"result : "<<result<<endl;
calc.setstate(new dividestate);
result=calc.evaluate("1-2+4*3-6/2+8*3-2*70/10");
cout<<"result : "<<result<<endl;
return 0;
}
```

Output

```
dividetstate : 1+3/3*2-2+6/2/3-2
multiplierstate : 1+1*2-2+1-2
substractstate : 1+2-2+1-2
addstate : 1+0+-1
result : 0
dividetstate : 1-2+4*3-6/2+8*3-2*70/10
multiplierstate : 1-2+4*3-3+8*3-2*7
substractstate : 1-2+12-3+24-14
addstate : -1+9+10
result : 18
```

- Observer

When a subject is observed by many observers, it leads to data view/controller model and forms an observer pattern. Subject which keeps the data keeps observers registered with it, data changes in the subject triggers the event sent to all the observers who can update their states. One of the observers may again make some tuning in the subject leading to data change in the subject, which makes other observers get informed. For example in an auction, an auctioneer is a subject whereas bidders are observers. Auctioneer starts with an initial value and lets the bidders (observers/view) know. An observer (controller) makes a bid and Subject (model/data) changes the auction rate and announces it again to all the observers(viewers). Typically any general election happens and many observers get placed at different stations from many different countries.



© www.minhinc.com

```
 +--------------+ +-----------------+ +-------------+
```

Code

```
#include <iostream>
#include <list>

using namespace std;


class bidder;
class subject{
public:
int state;
list<bidder*> observerlist;
void attach(bidder *observerp){observerlist.push_back(observerp);cout<<"new observer attached"<<endl;}
void detach(bidder *observerp){observerlist.remove(observerp);cout<<"an observer detached"<<endl;}
void notify();
void update(int statep){
state=statep;
cout<<"subject::update current bid value is Rs "<<state<<endl;
notify();
}
};

class bidder{ /*observer*/
subject* _subject;
public:
bidder(subject *subjectp):_subject(subjectp){_subject->attach(this);}
virtual void update(int value)=0;
};

void subject::notify(){
list<bidder*>::iterator it;
for (it=observerlist.begin();it!=observerlist.end();it++)
(*it)->update(state);
}

class indianbidder:public bidder{
int bidvalue;
public:
indianbidder(subject *subjectp,int statep):bidder(subjectp),bidvalue(statep){}
void update(int statep){
cout<<"indiabidder::update current bid value is Rs "<<statep<<" , my bid value is Rs "<<bidvalue<<endl;
}
};

class dollarbidder:public bidder{
int bidvalue;
public:
dollarbidder(subject *subjectp,int statep):bidder(subjectp),bidvalue(statep){}
void update(int statep){
cout<<"dollarbidder::update current bid value is Rs "<<statep<<" , my bid value is $ "<<bidvalue<<endl;
}
};


int main(void){
subject subjecti;
indianbidder ibidderi(&subjecti,100);
dollarbidder dbidderi(&subjecti,10);
subjecti.update(10);
return 0;
}
```

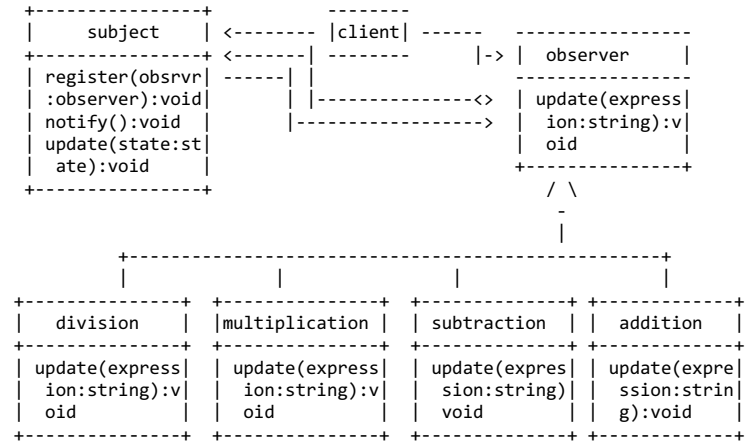
Output

```
new observer attached
new observer attached
subject::update current bid value is Rs 10
indiabidder::update current bid value is Rs 10 , my bid value is Rs 100
dollarbidder::update current bid value is Rs 10 , my bid value is $ 10
```

Article

https://www.codeguru.com/cpp/g-m/drawing-3d-opengl-graphics-on-google-maps.html

Arithmetic expression solution with observer:



Code

```
#include <iostream>
#include <list>
#include <regex>

using namespace std;
class subjectee;
class observer {
public:
subjectee *_subjectee;
observer(subjectee *subjecteep):_subjectee(subjecteep){}
virtual void update(string)=0;
};

class subjectee{
list<observer*> observerlist;
string data;
public:
char lastobserveroperator;
void attach(observer *observerp){observerlist.push_back(observerp);}
void update(const string& mepp,char lastobserveroperatorp){data=mepp;lastobserveroperator=lastobserveroperatorp;notify();}
void notify(){
list<observer*>::iterator it;
for(it=observerlist.begin();it!=observerlist.end();it++) (*it)->update(data);
}
string value(){return data;}
~subjectee(){
list<observer*>::iterator it;
for(it=observerlist.begin();it!=observerlist.end();it++) delete (*it);
}
};

class addobserver:public observer {
public:
using observer::observer;
void update(string mep){
if (_subjectee->lastobserveroperator=='-'){
smatch sm;
```

```
cout<<"addobserver : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
_subjectee->update(mep,'+');
}
}
};

class substractobserver:public observer {
public:
using observer::observer;
void update(string mep){
if (_subjectee->lastobserveroperator=='*'){
smatch sm;
cout<<"substractobserver : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\-(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
_subjectee->update(mep,'-');
}
}
};

class multiplierobserver:public observer {
public:
using observer::observer;
void update(string mep){
if (_subjectee->lastobserveroperator=='/'){
smatch sm;
cout<<"multiplierobserver : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
_subjectee->update(mep,'*');
}
}
};

class divideobserver:public observer {
public:
using observer::observer;
void update(string mep){
if (_subjectee->lastobserveroperator=='\0'){
smatch sm;
cout<<"divideobserver : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
_subjectee->update(mep,'/');
}
}
};

int main(void){
subjectee subjecteei;
subjecteei.attach(new divideobserver(&subjecteei));
subjecteei.attach(new multiplierobserver(&subjecteei));
subjecteei.attach(new substractobserver(&subjecteei));
subjecteei.attach(new addobserver(&subjecteei));
subjecteei.update("1+3/3*2-2+6/2/3-2",'\0');
cout<<"result : "<<subjecteei.value()<<endl;
subjecteei.update("1-2+4*3-6/2+8*3-2*70/10",'\0');
cout<<"result : "<<subjecteei.value()<<endl;
return 0;
}
```

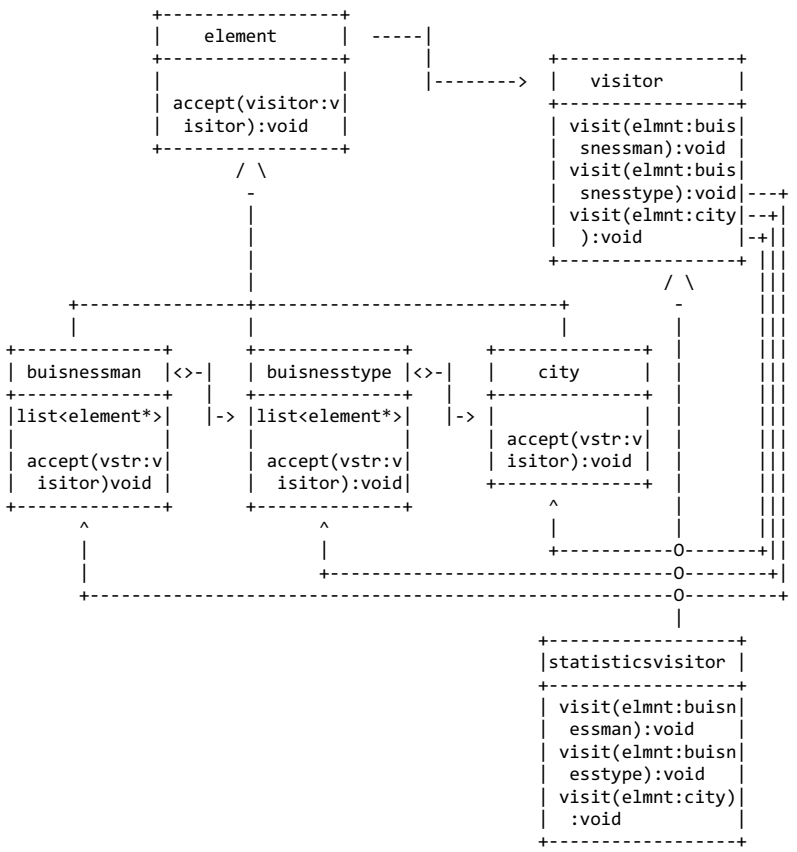
Output

```
divideobserver : 1+3/3*2-2+6/2/3-2
multiplierobserver : 1+1*2-2+1-2
substractobserver : 1+2-2+1-2
addobserver : 1+0+-1
result : 0
divideobserver : 1-2+4*3-6/2+8*3-2*70/10
multiplierobserver : 1-2+4*3-3+8*3-2*7
substractobserver : 1-2+12-3+24-14
addobserver : -1+9+10
result : 18
```

- Visitor

A collection of data types generally supports similar kind of operations. If a new operation is to be supported in a collection it would violate design principle if collection class adds new operations and it would be required at each collection classes. Better solution is to let a separate dedicated interface perform the task. Every subclass of an interface adds a new operation; this involves adding operation to a class at run time. This new interface is known as visitor pattern because it visits collection class in order to provide a new operation.
For example, a data container abstract class provides data sorting operation but does not provide list size and complexity calculation operation. A visitor class can have sub class statistics which calculates size and complexities of the member data respectively. Collection class can have a method that accepts the visitor classes and the calling method passes the call to visitor for specific operation in visitor implementation to take place.

Within a course of time many visitors may visit a province and write about its socio, economic, and political status. They collect these information from the province resources only.



Code

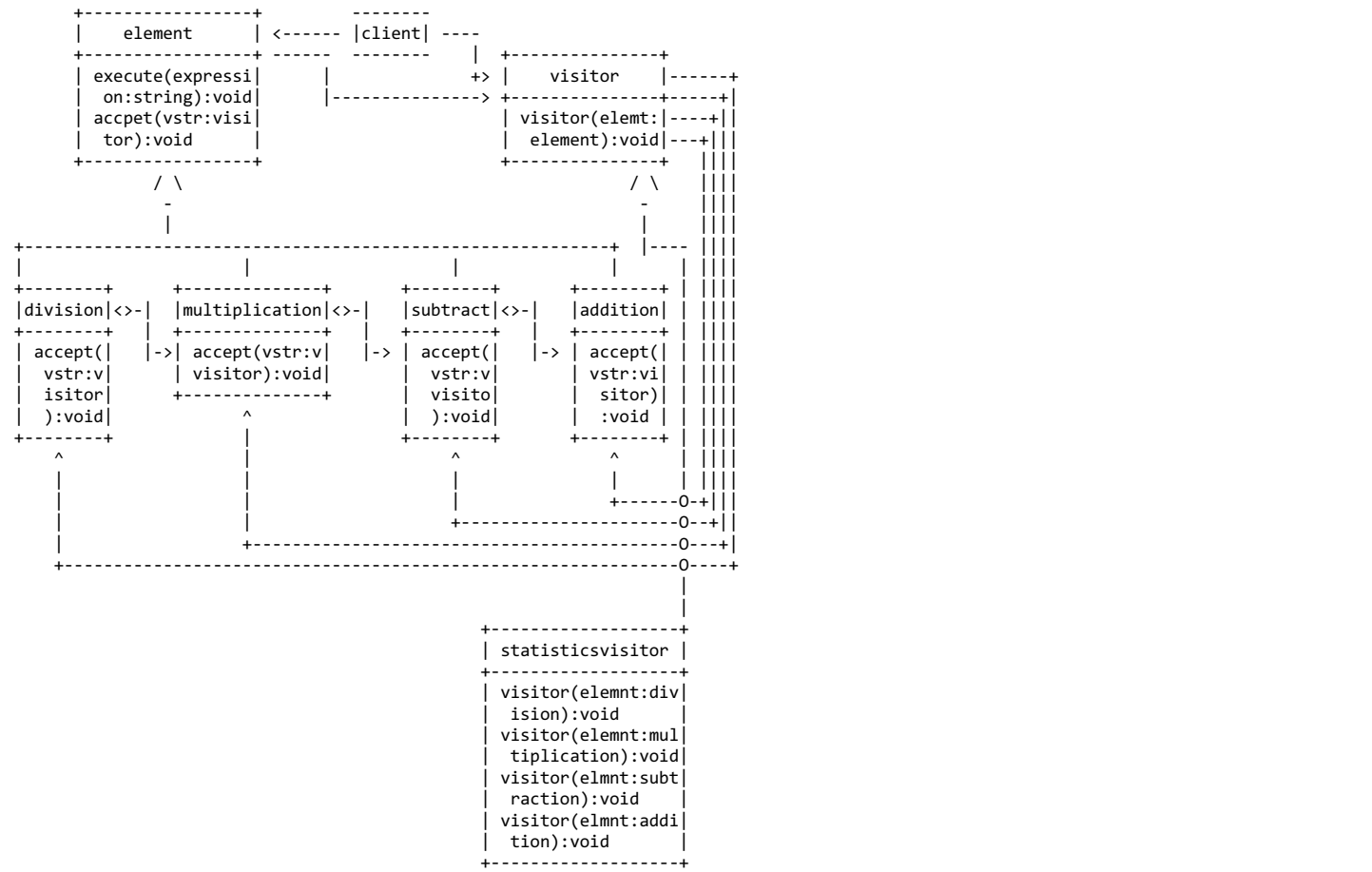
```
#include <iostream>
#include <list>

using namespace std;
class businessman;
class businesstype;
class businesscity;
class visitor{
public:
virtual void visit(businessman*)=0;
virtual void visit(businesstype*)=0;
virtual void visit(businesscity*)=0;
};

class element{
public:
virtual const string& name()=0;
virtual void accept(visitor*)=0;
};

class businessman:public element{
string _name;
public:
list<element*> businesstypelist;
businessman(const string& namep,const list<element*>& businesstypelistp):_name(namep),businesstypelist(businesstypelistp){};
const string& name(){return _name;}
void accept(visitor* v){
v->visit(this);
}
};

class businesstype:public element{
string _name;
public:
list<element*> businesscitylist;
businesstype(const string& namep,const list<element*>& businesscitylistp):_name(namep),businesscitylist(businesscitylistp){};
const string& name(){return _name;}
void accept(visitor* v){
v->visit(this);
}
};

class businesscity:public element{
```

```
string _name;
public:
businesscity(const string& namep):_name(namep){}
const string& name(){return _name;}
void accept(visitor* v){
v->visit(this);
}
};


class statisticalvisitor:public visitor{
void visit(businessman *elementp){
list<element*>::const_iterator it;
cout<<"businessman name: "<<elementp->name()<<endl;
for(it=elementp->businesstypelist.begin();it!=elementp->businesstypelist.end();it++)
(*it)->accept(this);
}
void visit(businesstype *elementp){
list<element*>::const_iterator it;
cout<<" businessman type: "<<elementp->name()<<endl;
for(it=elementp->businesscitylist.begin();it!=elementp->businesscitylist.end();it++)
(*it)->accept(this);
}
void visit(businesscity *elementp){
cout<<"  businesscity name: "<<elementp->name()<<endl;
}
};


int main(void){
element *elementi=new businessman("one",list<element*>{new businesstype("hardware",list<element*>{new businesscity("mangalore"),new
businesscity("bombay")}),new businesstype("software",list<element*>{new businesscity("chennai"),new businesscity("hamburg")})});
elementi->accept(new statisticalvisitor);
return 0;
}
```

Output

```
businessman name: one
 businessman type: hardware
  businesscity name: mangalore
  businesscity name: bombay
 businessman type: software
  businesscity name: chennai
  businesscity name: hamburg
```

Arithmetic expression solution through this pattern:



Code

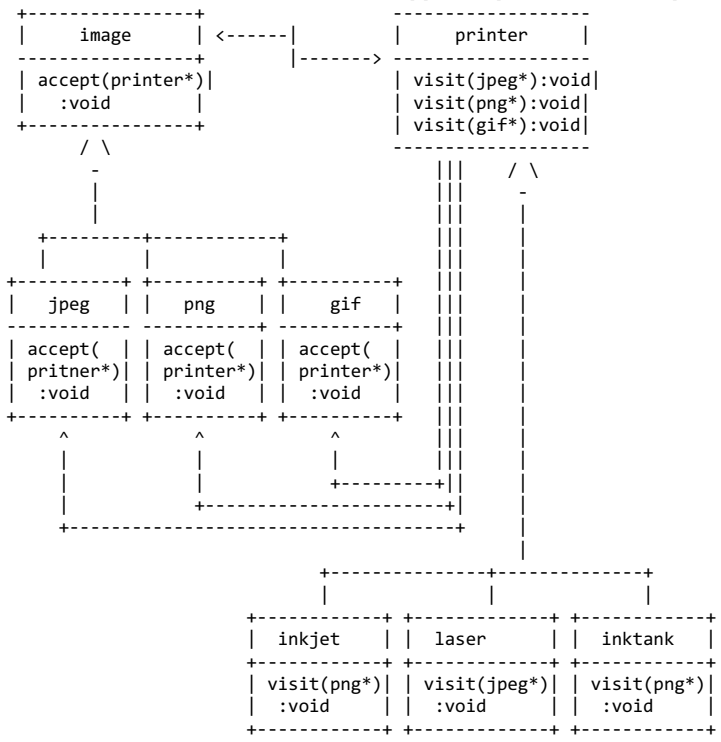
```
#include <iostream>
#include <regex>

using namespace std;
class addoperation;
class substractoperation;
class multiplieroperation;
class divideoperation;
class visitor {
public:
virtual void visit(addoperation*,string&)=0;
virtual void visit(substractoperation*,string&)=0;
virtual void visit(multiplieroperation*,string&)=0;
virtual void visit(divideoperation*,string&)=0;
};
class operation{
public:
virtual void accept(visitor *visitorp,string&)=0;
virtual ~operation(){};
};

class addoperation:public operation {
public:
void accept(visitor *visitorp,string& mep){
visitorp->visit(this,mep);
}
};

class substractoperation:public operation {
public:
operation *nextoperation;
substractoperation(operation *nextoperationp):nextoperation(nextoperationp){}
void accept(visitor *visitorp,string& mep){
visitorp->visit(this,mep);
}
~substractoperation(){delete nextoperation;}
};

class multiplieroperation:public operation {
public:
operation *nextoperation;
multiplieroperation(operation *nextoperationp):nextoperation(nextoperationp){}
void accept(visitor *visitorp,string& mep){
visitorp->visit(this,mep);
}
~multiplieroperation(){delete nextoperation;}
};

class divideoperation:public operation {
public:
operation *nextoperation;
divideoperation(operation *nextoperationp):nextoperation(nextoperationp){}
void accept(visitor *visitorp,string& mep){
visitorp->visit(this,mep);
}
~divideoperation(){delete nextoperation;}
};

class statisticalvisitor:public visitor{
public:
void visit(addoperation* addoperationp,string& mep){
smatch sm;
cout<<"addstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
}
void visit(substractoperation* substracteroperationp,string& mep){
smatch sm;
cout<<"substracterstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\-(\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
substracteroperationp->nextoperation->accept(this,mep);
}
void visit(multiplieroperation* multiplieroperationp,string& mep){
smatch sm;
cout<<"multiplierstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
multiplieroperationp->nextoperation->accept(this,mep);
}
void visit(divideoperation* divideoperationp,string& mep){
smatch sm;
cout<<"dividetstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
divideoperationp->nextoperation->accept(this,mep);
}
};

int main(void){
```

```
string me("1+3/3*2-2+6/2/3-2");
statisticalvisitor statisticalvisitori;
operation *operationi=new divideoperation(new multiplieroperation(new substractoperation(new addoperation)));
operationi->accept(&statisticalvisitori,me);
cout<<"result : "<<me<<endl;
me="1-2+4*3-6/2+8*3-2*70/10";
operationi->accept(&statisticalvisitori,me);
cout<<"result : "<<me<<endl;
return 0;
}
```

Output

```
dividetstate : 1+3/3*2-2+6/2/3-2
multiplierstate : 1+1*2-2+1-2
substracterstate : 1+2-2+1-2
addstate : 1+0+-1
result : 0
dividetstate : 1-2+4*3-6/2+8*3-2*70/10
multiplierstate : 1-2+4*3-3+8*3-2*7
substracterstate : 1-2+12-3+24-14
addstate : -1+9+10
result : 18
```

Priniting various images on various supported printers. Dual dispatching.




```
#include <iostream>

using namespace std;
class jpeg; class png; class gif;
class printer{
public:
virtual void visit(jpeg*){}
virtual void visit(png*){}
virtual void visit(gif*){}
};
class image{
public:
virtual void accept(printer*)=0;
};
class jpeg:public image{
public:
void accept(printer* visitorp){
visitorp->visit(this);
}
};
class png:public image{
public:
void accept(printer* visitorp){
visitorp->visit(this);
}
};
class gif:public image{
public:
void accept(printer* visitorp){
visitorp->visit(this);
}
};
class inkjet:public printer{
public:
```

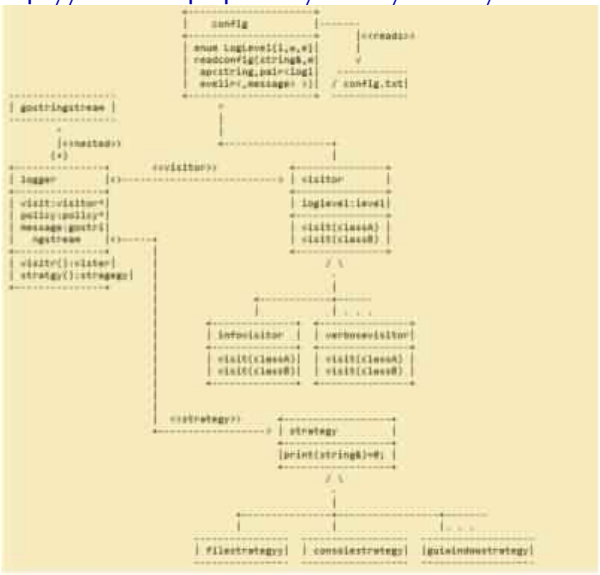
```
void visit(png *pngp){
cout<<"inkjet printing png"<<endl;
}
};
class laser:public printer{
public:
void visit(jpeg *jpeg){
cout<<"laser printing jpeg"<<endl;
}
};
class inktank:public printer{
public:
void visit(png *pngp){
cout<<"pngtank printing png"<<endl;
}
void visit(gif *gifp){
cout<<"inktank printing gif"<<endl;
}
};

int main(void){
int i,j;
image *imagearr[]={new jpeg,new png,new gif};
printer *printerarr[]={new inkjet,new laser,new inktank};
for (i=0;i<3;i++)
 for (j=0;j<3;j++)
  imagearr[i]->accept(printerarr[j]);
for (i=0;i<3;i++){
 delete imagearr[i];
 delete printerarr[i];
}
return 0;
}
```

Output

```
laser printing jpeg
inkjet printing png
pngtank printing png
inktank printing gif
```

Article

https://www.codeproject.com/Articles/869923/Class-Level-Generic-Logger



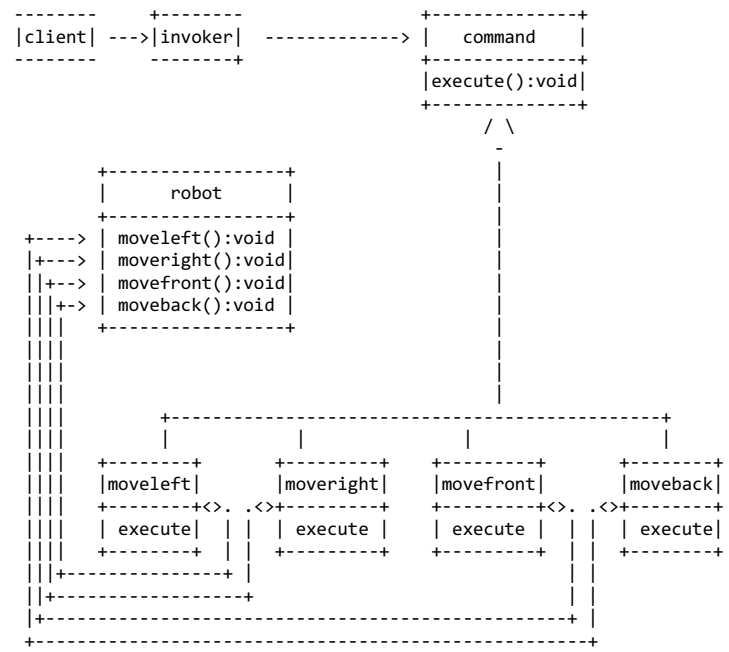
Day 4 Morning

7. GOF Behavioural Design Pattern - 2

- Non recursive calls, cont..
  - Directy Method call
  - Command (subclassed) pattern
  - Itertor pattern
  - Mediator pattern
  - Memento pattern
  - Strategy pattern
  - Template method pattern

Direct call patterns
As per the name a direct method calls others directly in non-recursive way and in non callback way.

- Command (subclassed)
In this approach action taken by subject class is given separate object identity. Each action (command) subclass actually calls subject's respective action. Program needs to mix their code with the command abstract class with an execute method called against it and depending on the subclass the object actual action would take place.



Code

```
#include <iostream>
#include <list>

using namespace std;
class robot {
public:
void left(){ cout<<"robot move left"<<endl; }
void right(){ cout<<"robot move right"<<endl; }
void front(){ cout<<"robot move front"<<endl; }
void back() { cout<<"robot move back"<<endl; }
};

class command {
public:
robot* rbt;
command(robot* rbtp):rbt(rbtp){}
virtual void execute()=0;
};

class moveleft:public command{
public:
using command::command;
void execute(){rbt->left();}
};

class moveright:public command{
public:
using command::command;
void execute(){rbt->right();}
```

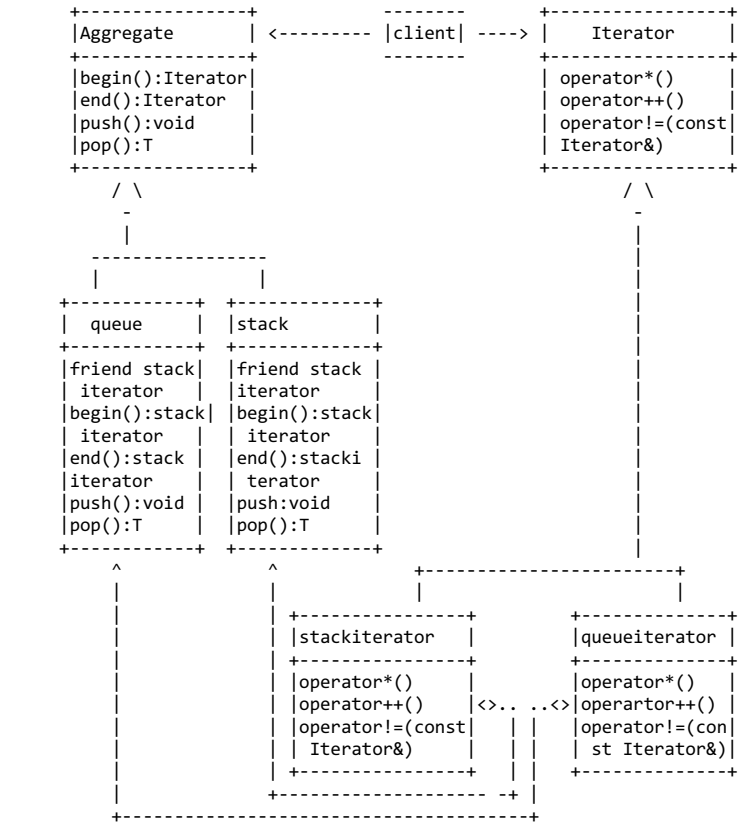
```
};

class movefront:public command{
public:
using command::command;
void execute(){rbt->front();}
};

class moveback:public command{
public:
using command::command;
void execute(){rbt->back();}
};

class invoker{
public:
void action(const list<command*>& commandlistp) {
list<command*>::const_iterator it;
for (it=commandlistp.begin();it!=commandlistp.end();it++)
(*it)->execute();
}
};

int main(void){
invoker invokeri;
robot roboti;
invokeri.action(list<command*>{new moveleft(&roboti),new movefront(&roboti),new moveright(&roboti),new moveback(&roboti)});
return 0;
}
```

Output

```
robot move left
robot move front
robot move right
robot move back
```

- Iterator

In a collection class when each element in the collection (i.e. array, list, tree etc) needs to be accessed sequentially and in a collection type independent manner, an iterator pattern is used. The interfaces provided through the pattern is the same for all collection classes, it can be array list tree or any other, and it is independent from the class internalrepresentation. For example a list user should be able to iterate the list in same manner as an array user. In CD player, user does not need to worry about what format songs are stored, he just presses next button in order to go to the next song.



Code

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

```
using namespace std;
class iteratorb{
public:
virtual int& operator*() const=0;
virtual iteratorb& operator++()=0;
virtual bool operator!=(const iteratorb&) const=0;
};

class stack;
class stackiterator:public iteratorb{
stack *_cnt;
int index;
public:
stackiterator(stack *stackp,int indexp):_cnt(stackp),index(indexp){}
int& operator*() const;
iteratorb& operator++();
bool operator!=(const iteratorb&) const;
};


class stack {
int _vec[10];
int sp;
public:
friend class stackiterator;
stack(){sp=0;}
void push(int vp){if(sp<10) _vec[sp++]=vp; }
int pop(){if(sp) return _vec[--sp];return -1;}
stackiterator begin(){return stackiterator(this,0);}
stackiterator end(){return stackiterator(this,sp);}
};

int& stackiterator::operator*() const{
return _cnt->_vec[index];
}

iteratorb& stackiterator::operator++(){
if (index<10) ++index;
return *this;
}

bool stackiterator::operator!=(const iteratorb& iterp) const{
return index!=(static_cast<const stackiterator&>(iterp)).index;
}

class queue;
class queueiterator:public iteratorb{
queue *_cnt;
int index;
public:
queueiterator(queue *queuep,int indexp):_cnt(queuep),index(indexp){}
int& operator*() const;
iteratorb& operator++();
bool operator!=(const iteratorb&) const;
};


class queue {
int _vec[10];
int bindex,eindex;
public:
friend class queueiterator;
queue(){bindex=eindex=0;}
void push(int vp){ if (eindex<10) _vec[eindex++]=vp; }
int pop(){if(bindex!=eindex) return _vec[bindex++]; return -1;}
queueiterator begin(){return queueiterator(this,bindex);}
queueiterator end(){return queueiterator(this,eindex);}
};

int& queueiterator::operator*() const{
return _cnt->_vec[index];
}

iteratorb& queueiterator::operator++(){
if (index<_cnt->eindex) ++index;
return *this;
}

bool queueiterator::operator!=(const iteratorb& iterp) const{
return index!=(static_cast<const queueiterator&>(iterp)).index;
}

void printdata(const iteratorb& itbp,const iteratorb& itep){
while(itbp != itep) {
cout<<*itbp<<" ";
++const_cast<iteratorb&>(itbp);
}
cout<<endl;
}
```

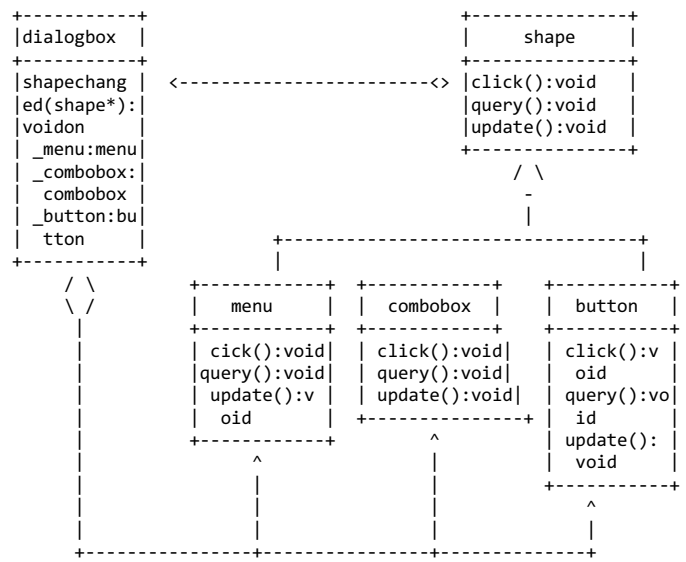
```
int main(void){
stack stacki;
for (int i=0;i<10;i++) {stacki.push(i);}
cout<<"----- stack data ------"<<endl;
printdata(stacki.begin(),stacki.end());
queue queuei;
for (int i=0;i<10;i++) {queuei.push(i);}
cout<<"----- queue data ------"<<endl;
printdata(queuei.begin(),queuei.end());
return 0;
}
```

Output

```
----- stack data ------
0 1 2 3 4 5 6 7 8 9
----- queue data ------
0 1 2 3 4 5 6 7 8 9
```

- Mediator

When a group of entities exist and change in ones state effect, other made entities are tightly coupled the same way. Adding or removing an entity will lead to change in the code of all other entities and when the group is big it's difficult to handle the situation. The Idea is to introduce a mediator and all the separate entity in the group would report to the mediator. This makes each entity in the group independent from each other and it's the mediator who decides updation among the entities. Telephone exchange(switch) connects N users, where connecting each other without a switch would be practically impossible. Head of a team is actually a mediator.




```
#include <iostream>
using namespace std;

class widget;
class mediator{
public:
enum widgetenum {menue,buttone,comboboxe};
widget *_widget[3];
mediator();
~mediator();
void widgetchanged(widget*);
widget* getwidget(widgetenum wep){
return _widget[wep];
}
};

class widget{
public:
mediator *_med;
widget(mediator *mediatorp):_med(mediatorp){}
virtual void signal()=0;
virtual void query()=0;
virtual void update()=0;
};
class menu:public widget{
public:
menu(mediator *mediatorp):widget(mediatorp){}
void signal(){
cout<<"menu::signal"<<endl;
_med->widgetchanged(this);
}
void query(){ cout<<" menu::query"<<endl; }
void update(){ cout<<" menu::update"<<endl; }
```

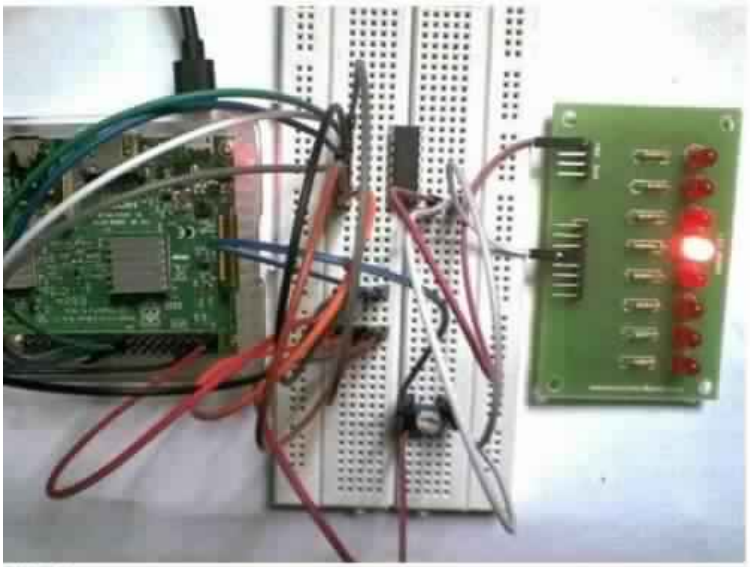
```
};

class button:public widget{
public:
button(mediator *mediatorp):widget(mediatorp){}
void signal(){
cout<<"button::signal"<<endl;
_med->widgetchanged(this);
}
void query(){ cout<<" button::query"<<endl; }
void update(){ cout<<" button::update"<<endl; }
};

class combobox:public widget{
public:
combobox(mediator *mediatorp):widget(mediatorp){}
void signal(){
cout<<"combobox::signal"<<endl;
_med->widgetchanged(this);
}
void query(){ cout<<" combobox::query"<<endl; }
void update(){ cout<<" combobox::update"<<endl; }
};

void mediator::widgetchanged(widget* w){
if(((menu*)w)==_widget[menue]){
 _widget[menue]->query();
 _widget[buttone]->update();
 _widget[comboboxe]->update();
}else if((menu*)w==_widget[buttone]){
_widget[buttone]->query();
_widget[comboboxe]->update();
}else if((combobox*)w==_widget[comboboxe]){
_widget[comboboxe]->query();
_widget[buttone]->update();
}
}

mediator::mediator(){
_widget[menue]=new menu(this);
_widget[buttone]=new button(this);
_widget[comboboxe]=new combobox(this);
}

mediator::~mediator(){
delete _widget[menue];
delete _widget[buttone];
delete _widget[comboboxe];
}

int main(void){
mediator mediatori;
mediatori.getwidget(mediator::menue)->signal();
mediatori.getwidget(mediator::buttone)->signal();
mediatori.getwidget(mediator::comboboxe)->signal();
return 0;
}
```

```
menu::signal
 menu::query
 button::update
 combobox::update
button::signal
 button::query
 combobox::update
combobox::signal
 combobox::query
 button::update
```

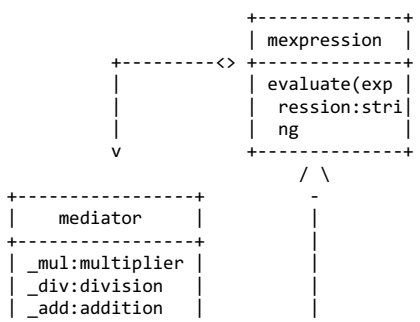
Article

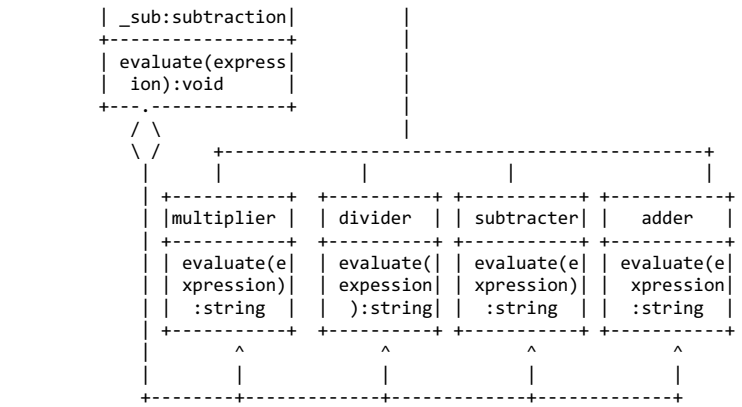
https://www.codeguru.com/cpp/g-m/drawing-3d-opengl-graphics-on-google-maps.html

https://www.codeguru.com/IoT/coding-sensors-on-the-rpi3.html



Arithmetic expression calculation:






```
#include <iostream>
#include <regex>

using namespace std;
class operation;
class mediator{
public:
enum operationenum{add,substract,multiply,divide};
operation *_operation[4];
mediator();
~mediator(){}
operation* getoperation(operationenum operationenump){return _operation[operationenump];}
void operationchanged(operation *opp);
string value();
};

class operation{
public:
string value;
mediator *_mediator;
operation(mediator *mediatorp):_mediator(mediatorp){}
virtual void signal(string)=0;
virtual string query()=0;
};

class addoperation:public operation{
public:
using operation::operation;
virtual void signal(string mep){
smatch sm;
cout<<"addstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\+(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))+stoi(sm.str(3)))+sm.str(4);
value=mep;
_mediator->operationchanged(this);
}
string query(){ return value; }
};

class substractoperation:public operation{
public:
using operation::operation;
virtual void signal(string mep){
smatch sm;
cout<<"substractstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\-(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))-stoi(sm.str(3)))+sm.str(4);
value=mep;
_mediator->operationchanged(this);
}
string query(){ return value; }
};

class multiplieroperation:public operation{
public:
using operation::operation;
virtual void signal(string mep){
smatch sm;
cout<<"multiplierstate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\*(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))*stoi(sm.str(3)))+sm.str(4);
value=mep;
_mediator->operationchanged(this);
}
string query(){ return value; }
};

class divideoperation:public operation{
public:
using operation::operation;
virtual void signal(string mep){
smatch sm;
```

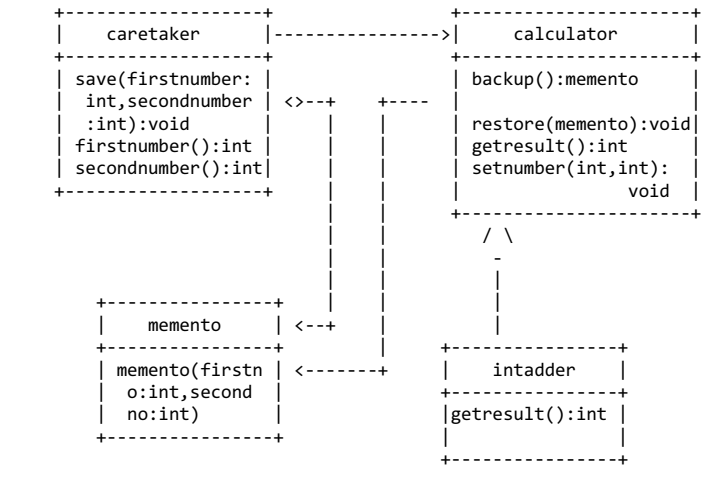
```
cout<<"dividestate : "<<mep<<endl;
while (regex_search(mep,sm,regex(R"((.*?)(-?\d+)\/(-?\d+)(.*))"))) mep=sm.str(1)+to_string(stoi(sm.str(2))/stoi(sm.str(3)))+sm.str(4);
value=mep;
_mediator->operationchanged(this);
}
string query(){ return value; }
};

void mediator::operationchanged(operation *opp){
if (((divideoperation*)opp)==_operation[divide])
 _operation[multiply]->signal(opp->query());
else if (((multiplieroperation*)opp)==_operation[multiply])
 _operation[substract]->signal(opp->query());
else if (((substractoperation*)opp)==_operation[substract])
 _operation[add]->signal(opp->query());
}

mediator::mediator(){
_operation[0]=new addoperation(this);
_operation[1]=new substractoperation(this);
_operation[2]=new multiplieroperation(this);
_operation[3]=new divideoperation(this);
}

string mediator::value(){
return _operation[add]->query();
}

int main(void){
mediator mediatori;
mediatori.getoperation(mediator::divide)->signal("1-2+4*3-6/2+8*3-2*70/10");
cout<<"result : "<<mediatori.value()<<endl;
mediatori.getoperation(mediator::divide)->signal("1+3/3*2-2+6/2/3-2");
cout<<"result : "<<mediatori.value()<<endl;
return 0;
}
```

Output

```
dividestate : 1-2+4*3-6/2+8*3-2*70/10
multiplierstate : 1-2+4*3-3+8*3-2*7
substractstate : 1-2+12-3+24-14
addstate : -1+9+10
result : 18
dividestate : 1+3/3*2-2+6/2/3-2
multiplierstate : 1+1*2-2+1-2
substractstate : 1+2-2+1-2
addstate : 1+0+-1
result : 0
```

- Memento

There are scenarios when a class (originator) changes its state and there has to be an option to bring the class to certain state that the class had been to in the past. It should support a kind of undo operation. A class can itself contain all the states it has happened to be in, in the past or a separate opaque class (memento) can be introduced to keep the state of the original class. A caretaker who handles the originator's class behavior keeps various states of the originator in opaque memento and when undo operation is required it sets the originator's state to a state stored in a particular memento object. For example a simple adder adds two numbers and gives the result. There can be a need that a user needs to see the previous calculations and in this case, the caretaker would let the originator to create many mementos of the past calculations and when an undo operation is required the previous mementos is set as the new state of the originator. Ministers are mementos for a kings actions and decisions and when required they make the king remember his past actions.



Code

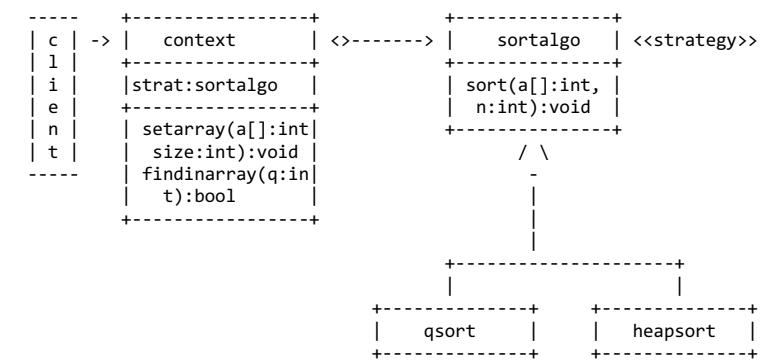
```
#include <iostream>
using namespace std;

class memento{
friend class calculator;
int left,right;
memento(int leftp,int rightp):left(leftp),right(rightp){}
};

class calculator {
int left,right;
public:
void setnumber(int leftp,int rightp){
left=leftp;
right=rightp;
}
int getleft(){return left;}
int getright(){return right;}
memento* backup(){return new memento(left,right);}
virtual int getresult()=0;
void restore(memento* mementop){
left=mementop->left;
right=mementop->right;
}
};


class adder:public calculator{ //originator
public:
adder(int leftp,int rightp){setnumber(leftp,rightp);}
int getresult(){return getleft()+getright();}
};


int main(void){ //caretaker
calculator *adderi=new adder(4,5);
memento *memento1=adderi->backup();
cout<<"adding 4, 5 : "<<adderi->getresult()<<endl;
adderi->setnumber(10,15);
memento *memento2=adderi->backup();
cout<<"adding 10, 15 : "<<adderi->getresult()<<endl;
adderi->restore(memento1);
cout<<"adding again 4, 5 : "<<adderi->getresult()<<endl;
adderi->restore(memento2);
cout<<"adding again 10, 15 : "<<adderi->getresult()<<endl;
return 0;
}
```

Output

```
adding 4, 5 : 9
adding 10, 15 : 25
adding again 4, 5 : 9
adding again 10, 15 : 25
```

- Strategy

The behavior of a class may contain both static and changeable behavior. Instead of hard-coding the changeable behavior it keeps abstract data type followed by getting the implementation of abstract at runtime. This kind of abstract data type is addressed through a strategy pattern. Abstract data type is then sub-classed into various implementations which are passed to the main class at run time making the possibility of changing the behavior at run time. i.e. an event manager when it receives an event it passes it through an algorithm which parses the event structure and takes necessary action, i.e. logging it, raises alarm etc. The algorithm can then be placed in a strategy abstract type so that it may vary and the event manager code remain unchanged leading to a new and better algorithm run in the fashion. A person plans (makes strategy) for his kid's life that at a particular age the kid has to go to school, then take a job there after and then marry at last; but he does not know the names.



```
                                | sort(a[]:int,|     | sort(a[]:int,|
                                |   n:int):void|     |  n:int):void |
                                +--------------+     +--------------+
```

```
#include <iostream>

using namespace std;
class sortalgo{
public:
virtual void sort(int a[],int lowindex ,int highindex)=0;
};

class context{
sortalgo *strategy;
int *array;
int arraysize;
bool sortedb;
public:
context():sortedb(false){}
void setstrategy(sortalgo *sortalgop){strategy=sortalgop;}
void setarray(int a[],int sizep){
array=a;
arraysize=sizep;
sortedb=false;
}
bool search(int search){
bool result=true;
int first,last,middle;
if (!sortedb) {
 strategy->sort(array,0,arraysize-1);
 sortedb=true;
}
first = 0; last = arraysize - 1; middle = (first+last)/2;
while (first <= last) {
 if (array[middle] < search) first = middle + 1;
 else if (array[middle] == search) break;
      else
       last = middle - 1;
 middle = (first + last)/2;
}
if (first > last) result=false;
return result;
}
};

class quicksort:public sortalgo{
public:
void sort(int array[],int m,int n){
int i,j,t;
if(m<n) {
i=m;j=n+1;
do {
 do {
  i=i+1;
 }while(i<=n && array[i]<array[m]);
 do {
  j=j-1;
 }while(j>=m && array[j]>array[m]);
if(i<j){t=array[i];array[i]=array[j];array[j]=t;}
}while(i<j);
t=array[j];array[j]=array[m];array[m]=t;
sort(array,m,j-1);
sort(array,j+1,n);
}
}
};

class heapsort:public sortalgo{
public:
void sort(int a[],int m,int n){
int i,t;
n=n+1;
heapify(a,n);
for (i=n-1;i>0;i--) {
 t = a[0]; a[0] = a[i]; a[i] = t; adjust(a,i);
}
}
void heapify(int a[],int n) {
int k,i,j,item;
for (k=1;k<n;k++) {
 item = a[k]; i = k; j = (i-1)/2;
 while((i>0)&&(item>a[j])) {
  a[i] = a[j]; i = j; j = (i-1)/2;
 }
 a[i] = item;
}
}
void adjust(int a[],int n) {
int i,j=0,item;
item = a[j]; i = 2*j+1;
while(i<=n-1) {
```

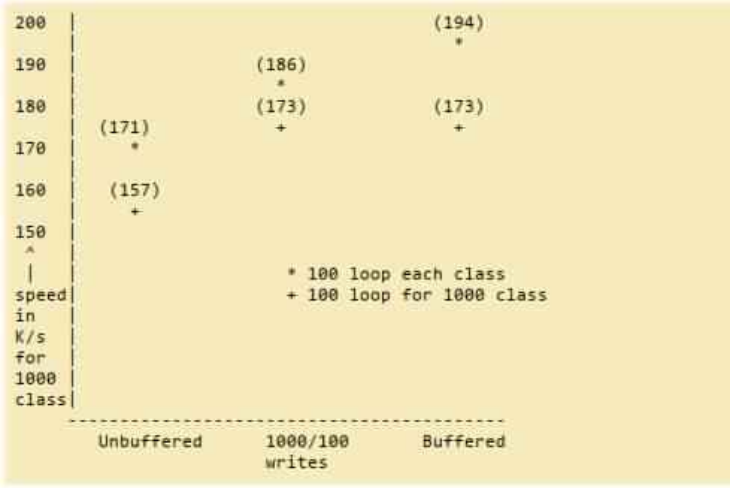
```
 if(i+1 <= n-1)
  if(a[i] <a[i+1])
   i++;
 if(item<a[i]) {
  a[j] = a[i]; j = i; i = 2*j+1;
 } else
  break;
}
a[j] = item;
}
};

ostream& operator<<(ostream& os, int arr[]){
for(int i=0;i<=16;i++) os<<arr[i]<<" ";
return os;
}

int main(void){
context contexti;

int arr[]={1,3,4,2,6,7,9,10,12,3,14,15,21,2,3,4,5};
sortalgo *qsorti=new quicksort;
contexti.setarray(arr,17);
contexti.setstrategy(qsorti);
cout<<"array for search through qsort : "<<arr<<endl;
cout<<"search 5 in array through qsort : "<<contexti.search(5)<<endl;
cout<<"search 8 in array through qsort : "<<contexti.search(8)<<endl;

int arr1[]={1,3,4,2,6,7,9,10,12,3,14,15,21,2,3,4,5};
sortalgo *heapsorti=new heapsort;
contexti.setarray(arr1,17);
contexti.setstrategy(heapsorti);
cout<<"array for search through heapsort : "<<arr1<<endl;
cout<<"search 5 in array through heapsort : "<<contexti.search(5)<<endl;
cout<<"search 8 in array through heapsort : "<<contexti.search(8)<<endl;

delete qsorti;
delete heapsorti;
return 0;
}
```

Output

```
array for search through qsort : 1 3 4 2 6 7 9 10 12 3 14 15 21 2 3 4 5
search 5 in array through qsort : 1
search 8 in array through qsort : 0
array for search through heapsort : 1 3 4 2 6 7 9 10 12 3 14 15 21 2 3 4 5
search 5 in array through heapsort : 1
search 8 in array through heapsort : 0
```

Article

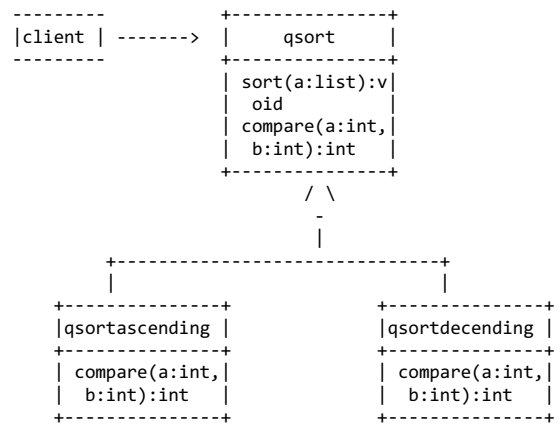
https://www.codeproject.com/Articles/869923/Class-Level-Generic-Logger



- Template method

As per the definition of template, a template draws an architecture of something and lets the implementor to implement the architecture from his perspective. A template is kind of a starting point. Similarly template method draws a layout of behaviors. A class containing the template method later sub-classes in order to fill the architecture into real examples. For example someone writes an algorithm to sort an array or list. Sorting can happen in at-least two ways either ascending or descending.

Rather than mentioning these in the algorithm he just writes the architecture and leaves a hook point where ascending and descending specific codes in a specific subclasses would make it two separate algorithms; ascending sorting and descending sorting. A person can work on one task, two or three, but if he wants to work on 100s of tasks, then either he cannot or he has to find common things among them so that they would differ in very few points.



Code

```
#include <iostream>
using namespace std;
class qsort{
public:
int* sort(int array[],int m,int n);
virtual bool compare(int,int)=0;
};

int* qsort::sort(int array[],int m,int n){
int i,j,t;
if(m<n) {
i=m;j=n+1;
do {
 do {
  i=i+1;
 }while(i<=n && compare(array[i],array[m]));
 do {
  j=j-1;
 }while(j>=m && compare(array[m],array[j]));
if(i<j){t=array[i];array[i]=array[j];array[j]=t;}
}while(i<j);
t=array[j];array[j]=array[m];array[m]=t;
sort(array,m,j-1);
sort(array,j+1,n);
}
return array;
}

class qsortascend:public qsort{
public:
bool compare(int a,int b){ return a<b; }
};

class qsortdescend:public qsort{
public:
bool compare(int a,int b){return a>b;}
};


ostream& operator<<(ostream& os, int arr[]){
for(int i=0;i<=16;i++) os<<arr[i]<<" ";
return os;
}

int main(void){
int arr[]={1,3,4,2,6,7,9,10,12,3,14,15,21,2,3,4,5};
cout<<"Array to sort : "<<arr<<endl;
qsortascend sortascend;
cout<<"------- Ascending sort : --------"<<endl<<sortascend.sort(arr,0,16)<<endl;
qsortdescend sortdescend;
cout<<"------- Descending sort : --------"<<endl<<sortdescend.sort(arr,0,16)<<endl;
return 0;
}
```

Output

```
Array to sort : 1 3 4 2 6 7 9 10 12 3 14 15 21 2 3 4 5
------- Ascending sort : --------
1 2 2 3 3 3 4 4 4 5 6 7 9 10 12 14 15 21
------- Descending sort : --------
21 15 14 12 10 9 7 6 5 4 4 3 3 3 2 2 1
```

© www.minhinc.com

©www.minhinc.com