

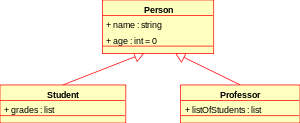
We can tell the relationships between classes by using various styles of connecting lines and arrowheads:

1. **Association:** A simple dependency between two classes where neither owns the other, indicated as a solid line. Association can be directional, shown with an open arrowhead (">").
2. **Aggregation:** A **has-a**, or whole/part, relationship where neither class owns the other, indicated as a solid line with a hollow diamond.
3. **Composition:** A **has-a** relationship where the lifetime of the part is managed by the whole. This is shown as a solid line with a filled diamond.
4. **Generalization:** A subclass relationship between classes, represented as a hollow triangle arrowhead.

* [**Association**](https://www.uml-diagrams.org/association.html)**:**associations are a relationship between two classes at the class level. That is, one class keeps a direct or indirect “link” to the associated class as a member.

class **Doctor**   
{  
private:  
 vector<**Patient** \*> m\_patient{};  
public:  
 Doctor();  
 void addPatient(**Patient** \*pat);  
};  
   
class **Patient**  
{  
private:  
 vector<**Doctor** \*> m\_doctor{}; // so that we can use it here  
   
 // make addDoctor private because we don't want the public to use it.  
 // They should use Doctor::addPatient() instead, which is publicly exposed  
 void addDoctor(**Doctor** \*doc)  
 {  
 m\_doctor.push\_back(doc);  
 }  
   
public:  
 Patient();// We'll friend Doctor::addPatient() so it can access the private function Patient::addDoctor()  
 friend void Doctor::addPatient(Patient \*pat);  
};

* **Inheritance**



class **Person**  
{  
public:  
private:   
};class **Student**: public **Person**  
{  
 ...  
};class **Professor**: public **Person**  
{  
 ...  
};

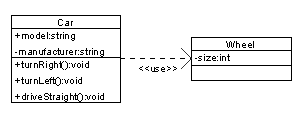
* **Realization/Implementation**

https://miro.medium.com/max/270/0*wzjvTQfHx3cV3_TA.png

***Interface****SiteSearch is****realized****(implemented) by SearchService*

class **Shape** // An interface class only contains pure virtual functions  
{  
 public:  
 virtual ~shape();  
 virtual void move\_x(int x) = 0;  
 virtual void move\_y(int y) = 0;  
 virtual void draw() = 0;  
};class **Line** : public **Shape**  
{  
 public:  
 virtual ~line();  
 virtual void move\_x(int x); // implements move\_x  
 virtual void move\_y(int y); // implements move\_y  
 virtual void draw(); // implements draw  
 private:  
 point end\_point\_1, end\_point\_2;  
};

* **Dependency:**Dependencies typically are not represented at the class level — that is, the object being depended on is not linked as a member. Rather, the object being depended on is typically instantiated as needed. e.g. Our classes that use std::cout to accomplish the task of printing something to the console.



class **Point**  
{  
private:  
double m\_x, m\_y;  
  
public:  
Point();  
friend std::ostream& operator<< (std::**ostream** &out, const Point &point);  
};  
  
std::ostream& operator<< (std::**ostream** &out, const Point &point)  
{  
// Since operator<< is a friend of the Point class, we can access Point's members directly.  
out << "Point(" << point.m\_x << ", " << point.m\_y;  
return out;  
}

* **Aggregation**1. When representing a software or database relationship, e.g. car model engine ENG01 is part of a car model CM01, as the engine, ENG01, may be also part of a different car model.2. When the container is destroyed, the contents are usually not destroyed, e.g. a professor has students, when the professor dies the students don’t die along with him or her.

https://miro.medium.com/max/300/0*tVXE2Ls6r1MTPoKW.png

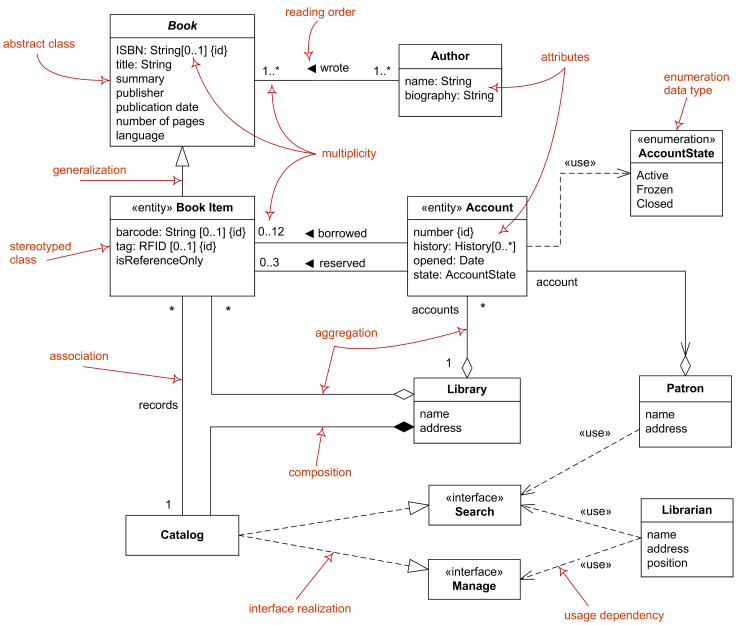
class **Class**  
{  
private:  
string m\_name;  
public:  
Class(string name)  
: m\_name(name)  
{}  
string getName() { return m\_name; }  
};  
  
class **Professor**  
{  
private:  
**Class** \*m\_class; // holds only one class for simplicity, but it could teach many classes  
public:  
Professor(**Class** \*class = nullptr)  
: m\_class(class)  
{}  
};

* The **Composition** is a strong aggregation.  
  1. It is a **binary**association  
  2. It is a *whole/part* relationship  
  3. A part could be included in *at most one* composite (whole) at a time  
  4. If a composite (whole) is deleted, all of its composite parts are “normally” deleted with it.

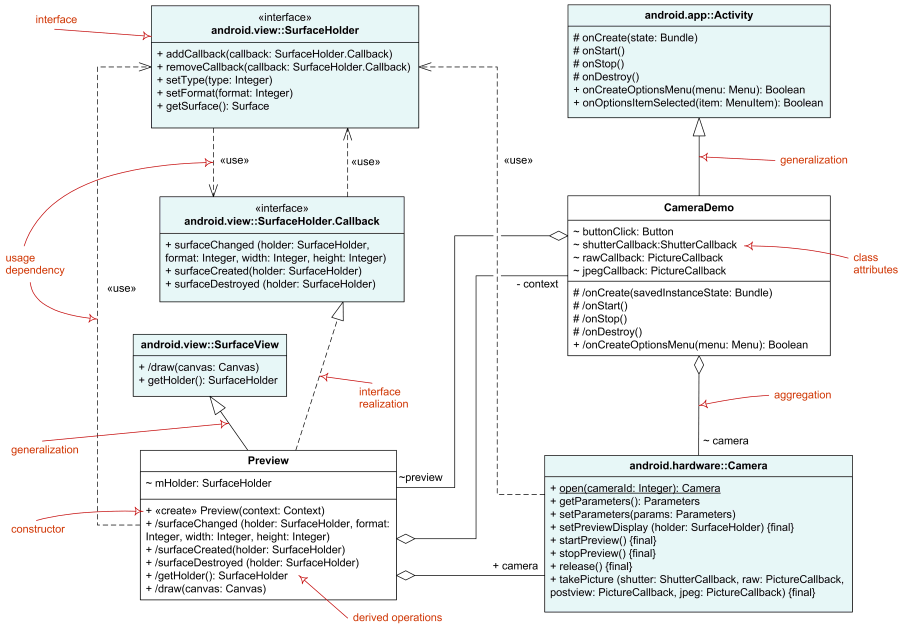
https://miro.medium.com/proxy/0*ahmZl_DqjQ_11Xrw.png

class **Department**  
{  
private:  
 string name;  
public:  
 Department(){}  
};class **Hospital**{  
public:  
 Hospital(){}  
private:  
 vector<**Department\***> m\_departments;  
};

# Domain Model Diagram



# Diagram of Implementation Classes:



# Object Diagram

