Contents

[Contents 1](#_Toc515620528)

[1 Class Diagrams 2](#_Toc515620529)

[1.1 Active Classes 2](#_Toc515620530)

[1.1.1 A class representation 2](#_Toc515620531)

[1.1.2 Different visibility of the class can be represented as 2](#_Toc515620532)

[1.1.3 Different Parameter direction 3](#_Toc515620533)

[1.1.4 Different type of members in a class 3](#_Toc515620534)

[1.1.5 Different Multiplicity in a relation 3](#_Toc515620535)

[1.1.6 Class relationship 3](#_Toc515620536)

[1.1.7 Class relationship Examples 3](#_Toc515620537)

[1.1.7.1 Association 3](#_Toc515620538)

[1.1.7.2 Dependency 4](#_Toc515620539)

[1.1.7.3 Composition 5](#_Toc515620540)

[1.1.7.4 Aggregation 5](#_Toc515620541)

[1.1.7.5 Inheritance (Generalization) 6](#_Toc515620542)

[1.1.7.6 Class Template 7](#_Toc515620543)

[1.1.8 Association, Aggregation, and Composition 7](#_Toc515620544)

[2 Design Pattern 8](#_Toc515620545)

[2.1 Singleton Design Pattern 8](#_Toc515620546)

[2.2 Factory Method Design Pattern 8](#_Toc515620547)

[2.2.1 Definition 8](#_Toc515620548)

# Class Diagrams

Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.

Illustrate classes with rectangles divided into compartments. Place the **name of the class** in the **first partition** (centered, bolded, and capitalized), **list the attributes** in the **second partition** (left-aligned, not bolded, and lowercase), and write **operations** into the **third**.

## Active Classes

Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.



### A class representation

class Circle {

private:

double radius;

Point center;

public:

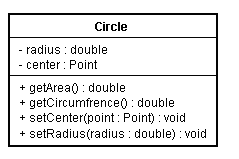
setRadius(double radius);

setCenter(Point center);

double getArea();

double getCircumfrence();

};



### Different visibility of the class can be represented as



### Different Parameter direction

**“in”** The parameter is an input parameter.

**“Inout”** The parameter is capable of both input and output.

**“Out”** The parameter is an output parameter.

### Different type of members in a class

* Static members are represented as underlined.
* Pure virtual functions are represented as italics.

### Different Multiplicity in a relation

* “0..1” No instances, or one instance (optional, may)
* “1” Exactly one instance
* “0..\* or \*” Zero or more instances
* “1..\*” One or more instances (at least one)

### Class relationship

In a system a class may be related to different classes, following are the different relationship.

* Association (knows a)
* Dependency (uses a)
* Composition (has a)
* Aggregation (has a)
* Inheritance (is a)
* Class template

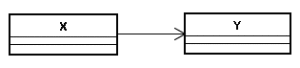
### Class relationship Examples

* Manager (is an) employee of XYZ limited corporation. **Inheritance (is a)**
* Manager (has a) swipe card to enter XYZ premises. **Association (knows a)**
* Manager has workers who work under him. **Aggregation (has a)**
* Manager has the responsibility of ensuring that the project is successful. **Composition (has a)**
* Manager's salary will be judged based on project success. **Composition (has a)**

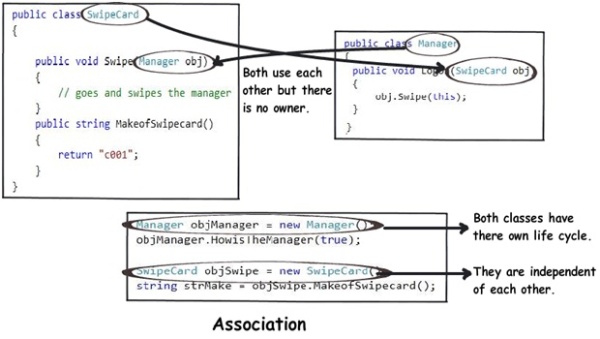
#### Association

Association relationship develops between objects when one object uses another object. Technically, it is a reference based relationship between two classes. One class holds a class level reference to the other class.

Representation:



Manager uses a swipe card to enter XYZ premises. In this requirement, the manager object and the swipe card object use each other but they have their own object life time. In other words, they can exist without each other. The most important point in this relationship is that there is no single owner.



The above diagram shows how the SwipeCard class uses the Manager class and the Manager class uses the SwipeCard class. You can also see how we can create objects of the Manager class and SwipeCard class independently and they can have their own object life time.

This relationship is called the “Association” relationship.

#### Dependency

One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class

Representation:

class_diagram_dependency

class GreetingSender

{

EmailSender \_emailSender;

void SendGreetings(EmailSender emailSender)

{

\_emailSender = emailSender;

//Send Greeting through Email

\_emailSender.SendEmail();

}

}

class EmailSender

{

public void SendEmail()

{

//Send Email

}

}

#### Composition

Composition is the stronger form of aggregation. Composition can occur when a class is a collection or container of other classes, but where the contained classes have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are also destroyed

Representation:

class_diagram_composition

The last two requirements are actually logically one. If you read closely, the requirements are as follows:

Manager has the responsibility of ensuring that the project is successful.

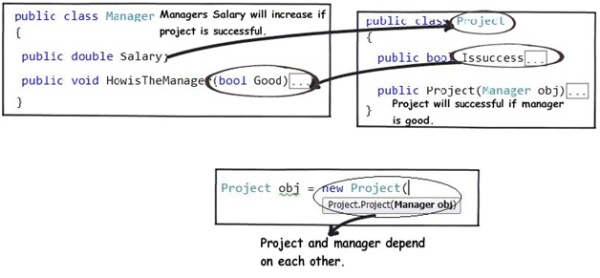
Manager's salary will be judged based on project success.

Below is the conclusion from analyzing the above requirements:

Manager and the project objects are dependent on each other.

The lifetimes of both the objects are the same. In other words, the project will not be successful if the manager is not good, and the manager will not get good increments if the project has issues.

Below is how the class formation will look like. You can also see that when I go to create the project object, it needs the manager object.

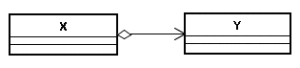


This relationship is termed as the composition relationship. In this relationship, both objects are heavily dependent on each other. In other words, if one goes for garbage collection the other also has to be garbage collected, or putting from a different perspective, the lifetime of the objects are the same. That’s why I have put in the heading “Death” relationship.

#### Aggregation

Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are not. You may have confusion between aggregation and association .Association differs from aggregation only in that it does not imply any containment.

Representation:

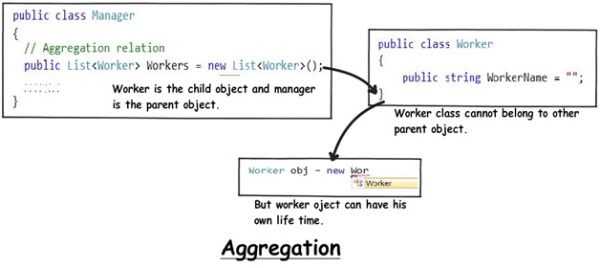


(Manager has workers who work under him) denotes the same type of relationship like association but with a difference that one of them is an owner. So as per the requirement, the Manager object will own Worker objects.

The child Worker objects can not belong to any other object. For instance, a Worker object cannot belong to a SwipeCard object.

But… the Worker object can have its own life time which is completely disconnected from the Manager object. Looking from a different perspective, it means that if the Manager object is deleted, the Worker object does not die.

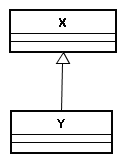
This relationship is termed as an “Aggregation” relationship.



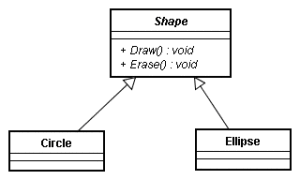
#### Inheritance (Generalization)

In Inheritance relationship a class is derived from another class. It is a “is a” relationship between two classes.

Representation:



Here X and Y are normal classes.

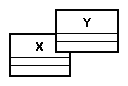


Here Shape is an abstract class that is why it is shown in Italics. Draw () and Erase () methods of Shape class is pure virtual function, so it is also shown as italics.

#### Class Template

Template class mean generic classes.Languages like C++, java, C# supports generic programming.

Representation:



template <class T>

class X {

...

...

...

};

X Y

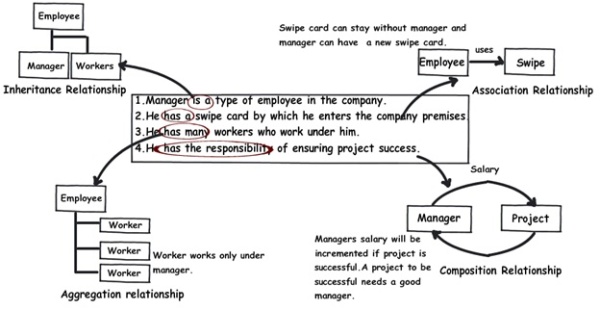
...

X<Y> a;

...

### Association, Aggregation, and Composition

Below is a visual representation of how the relationships have emerged from the requirements.



|  |  |  |  |
| --- | --- | --- | --- |
|  | **Association** | **Aggregation** | **Composition** |
| **Owner** | No owner | Single owner | Single owner |
| **Life time** | Have their own lifetime | Have their own lifetime | Owner's life time |
| **Child object** | Child objects all are independent | Child objects belong to a single parent | Child objects belong to a single parent |

# Design Pattern

## Singleton Design Pattern

// Static initializer based C++ implementation of

// singleton design pattern

class Singleton

{

private static Singleton obj = new Singleton();

private Singleton() {}

public static Singleton getInstance()

{

return obj;

}

};

## Factory Method Design Pattern

### Definition

Factory method is a ***creational design pattern***, i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.

The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library (explained below) in a way such that it doesn’t have tight coupling with the class hierarchy of the library.

***What is meant when we talk about library and clients?***

A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.