# Project Milestone #1: Domain Model (Conceptual UML Class Diagram)

In preparation for the class project, construct a conceptual UML Class Diagram (A conceptual model captures the important concepts and relationships in some domain. Concepts are represented by classes, while relationships are represented by associations) that models the Domain Concepts described in the [Course Project](#_Course_Project)overview (Users, Statuses, Feeds, etc.). Capture attributes that might exist in each class. You should lean towards associations (instead of attributes) for anything that is not a simple, primitive value. You do **NOT** need to include operations for classes. Capture the relationships between classes. Label the UML Class diagram correctly with appropriate symbols to represent **association, multiplicity/cardinality, composition, aggregation,** etc. as needed. Include a note for every class and complex relationship that needs explaining to describe what it is to another person that wants to review your diagram.

Use an appropriate software choice to create the diagram. For example, LucidChart, yUML, etc. We recommend using [LucidChart](https://www.lucidchart.com/" \t "_blank). It provides the correct symbols and styles we are looking for. You can sign up for an education account that will allow your account to have sufficient space to create documents for this class.  Here's a link to a video made by Lucid Software that shows how to create UML diagrams in LucidChart: [video](https://www.youtube.com/watch?v=UI6lqHOVHic).

For definitions of each term, refer to the Project Overview Documentation.

### **Submission:**

* Submit a pdf of your diagram on Canvas
* (Optionally) You may resubmit your class diagram with any corrections within 72 hours of getting the graded diagram back, for the possibility of getting half of the points you lost back.

### **Requirements:**

The following checklist of requirements will be used by the TA in reviewing your UML diagram.

* Have appropriate classes: User, Status, Feed, etc.
  + Classes can have attributes but favor associations for non-basic types (eg, int, string)
  + Anything that is not apparently clear has a note describing its purpose
* Has a relationship or class capturing the follows relationship.
* Has associations between classes that are related
  + Each association has a name (with a direction on that name label, if it is helpful)
  + Each association has multiplicities/cardinalities
* Includes at least one composition or aggregation relationship.
  + Each relationship uses the correct symbol
  + Each relationship uses the correct direction
* Is well organized, neat, and readable



Status

! Status is only included in user’s feed if status was posted at the time the user was following the author. Role labels.

Profile Page

default: Feed {time stamp}

newest to oldest

Login Page

Login/Register

View Followers

View Following

View Story

View Feed

View write post

Button to write a story is in all 4 views of user

Other user page

Following/Follow

View Followers

View Following

View Story

# 

# Course Project

## Domain Concepts

**Your application will be based on the following *domain concepts*:**

* **User**of the system has the following attributes
  + Name: the user’s real name (eg. Matt Pope).
  + Alias: the user’s login ID or handle (eg. matt).
  + Photo: a picture of the user.
* A**status** (ie message) is a publicly readable character limited length string, which can contain
  + User mentions which is a user alias preceded by the “@” symbol (eg, *@matt*)
  + URLs
* A user’s**story** is all of the statuses posted by that user.
* **Follows**is an asymmetrical relationship between users, meaning user A can follow user B without user B following user A. We refer to all of the users following user A as user A’s *followers*. We refer to all of the users A follows as A’s *following*.
* A user’s **feed** is all of the statuses posted by users he or she follows, sorted from newest to oldest. A detail to note here is that a status is only included in user’s feed if the status was posted at the time the user was following the author.

## User interface

**Here are a few general user interface considerations, spanning multiple application views.**

1. All of the main views are *paginated* lists of statuses or users. By paginated we mean that the content is loaded a “page” at a time. The application will add an additional page of content when the user of the app reaches the bottom of the page *or* using something like a “more” button in the user interface.
2. All displays of statuses turn user mentions (@...) and URLs into clickable links. A *user mention*links to a user story view (which page should facilitate access to that user’s statuses, followers and following).
3. All displays of statuses include the profile image of the author of the status
4. Across all views, actions that are impossible should be disabled or not included in the user interface at all. For example, *follow* or *unfollow* functionality only make sense when a user is logged in and depends on whether the currently logged in user is (or is not) already following a given user.
5. Errors should be handled and (when appropriate) communicated to users in a consistent way.

## Requirements

For milestone 2 to milestone 4 you will be implementing a core set of features for the application. More details on what is expected for each milestone will be shared in a separate document. Note that in the following when we state *“A signed in user can…”* we are implying that *only*a signed in user can perform that action. Similarly, when we state *“A user can…”* we mean that a user can perform that action whether or not they are signed in.

You may design your UI so it signs a user in first and, therefore, requires them to be signed in for all actions, or you may design your UI in a way that it support the actions that don't require a signed in user without requiring sign in. You must require sign in for the actions that are for a signed in user.

**Your application is to satisfy the following *user and session management* requirements:**

1. A new user can *sign up* for the service, specifying their name, alias, and password, and providing an image to upload (i.e., their profile photo). After signing up a user is automatically “signed in” and is redirected to their (at this point) blank feed.
   1. User passwords are to be stored as hashes
   2. User profile pictures are stored on AWS S3.
2. A user that has previously signed up for the service, can *sign in* (ie. authenticate) by supplying their alias and password. If the alias and password are correct the user is redirected to her or his feed.
   1. Authentication tokens are to expire after N minutes of inactivity.
3. A signed in user can *sign out* (ie log off) of the service.

**And *status* related requirements:**

1. A signed in user can *post a status*. The system will then add that newly created status to the feeds of all of the author’s followers.
2. A signed in user can view all statuses of all of his or her followers merged into one list, sorted from newest to oldest. We call this list of statuses the user’s **feed** and should be the default view of the application for a signed in user.
3. A user can view all statuses posted by a given user, sorted from newest to oldest. We call this list of statuses a user’s **story** and can be reached by clicking on a “mention” link (@…).

**And *following/followers* related requirements:**

1. A user can view all of the users*followed by* a particular user. Naturally this includes the ability for a signed in user to see all of the users she or he follows.
2. A user can view all of the users *following* a particular user. Again, this includes the ability for a signed in user to see all users following him or her.
3. A user can see a count of all of his or her followers and followees.
4. A signed in user can follow an unlimited number of other users, though a user cannot follow herself or himself or someone he or she is already following.
5. A signed in user can only unfollow users they are following. After user A unfollows user B, user B’s new statuses are no longer added to A’s feed, however all of B’s statuses that had previously been added to A’s feed remain in A’s feed.
6. A user can see a count of how many users are *followed by* a particular user. The same goes for the count of how many users are *following* that particular user. Naturally, these counts should change when users select to follow or unfollow another user.

**UML Association vs Aggregation vs Composition**

UML association

UML aggregation

UML composition

## **Association**

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).

We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association.

In UML, *cardinality* is represented by characters: “..1” (meaning that an instance of the first entity class can be associated with no more than one instance of the second class) or “..\*” (meaning that the first entity can be associated with an unlimited number of instances of the second class). A relationship’s *optionality* can be either “**0..**” (meaning that the relationship is optional) or “**1..**” (meaning that it is required).

The example indicates that a Student has one or more Instructors:

A single student can associate with multiple teachers:

Association multiplicity example 1

The example indicates that every Instructor has one or more Students:

Association multiplicity example 2

We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

Association multiplicity example 3

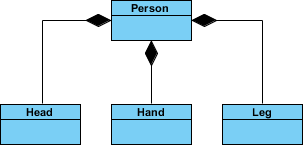
**Association vs Aggregation vs Composition**

Aggregation and Composition are subsets of **association** meaning they are specific cases of association. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

* **Aggregation** implies a relationship where the child can **exist independently** of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist. (Lose coupling)
* **Composition** implies a relationship where the child **cannot exist independent** of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

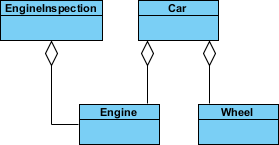
### **Composition Example:**

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



### **Aggregation Example:**

It's important to note that the aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.



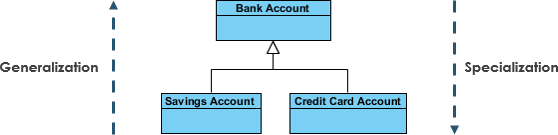
To sum it up association is a very generic term used to represent when one class used the functionalities provided by another class. We say it's a composition if one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object. If it can then it is called Aggregation.

## **Generalization vs Specialization**

Generalization is a mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. The commonality may be of attributes, behavior, or both. In other words, a superclass has the most general attributes, operations, and relationships that may be shared with subclasses. A subclass may have more specialized attributes and operations.

Specialization is the reverse process of Generalization means creating new sub-classes from an existing class.

For Example, a Bank Account is of two types - Savings Account and Credit Card Account. Savings Account and Credit Card Account inherit the common/ generalized properties like Account Number, Account Balance, etc. from a Bank Account and also have their specialized properties like unsettled payment etc.



## **Generalization vs Inheritance**

Generalization is the term that we use to denote abstraction of common properties into a base class in UML. The UML diagram's Generalization association is also known as Inheritance. When we implement Generalization in a programming language, it is often called Inheritance instead. Generalization and inheritance are the same. The terminology just differs depending on the context where it is being used.