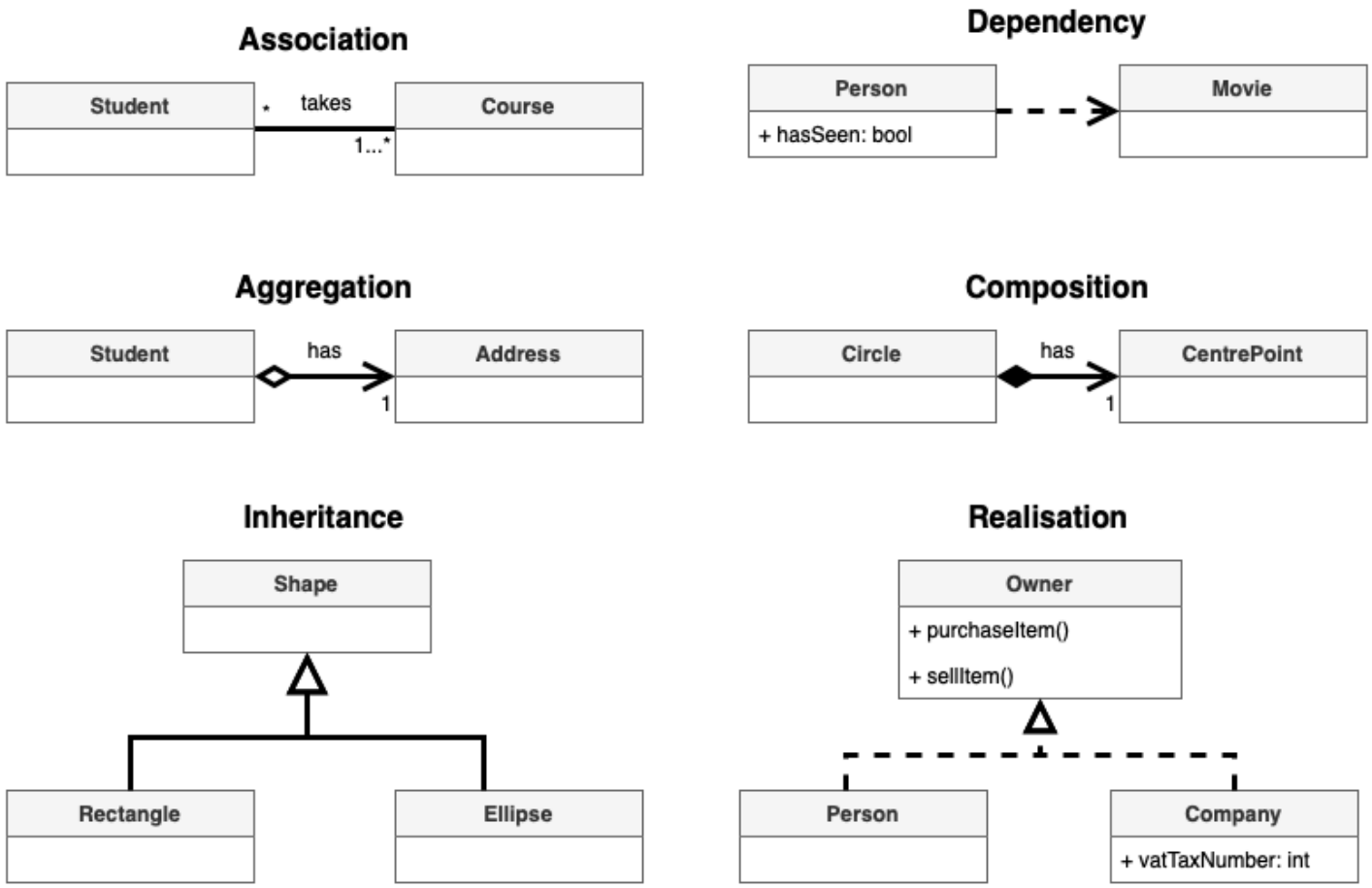
Software Engineering II Model Diagram Conventions

For this course, web application implementation is done in python & SQLAlchemy. This has implications on how we approach OOP design and thus the following are some conventions for UML modelling.

Types of Relationships



Association
Note association is a generalisation of composition and aggregation.

Aggregation vs Composition

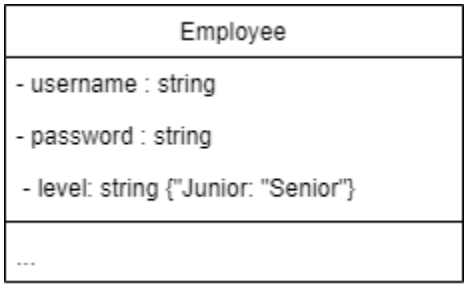
Class A is an aggregate of B if B can exist without an associated instance of A. If B can only exist as part of A then it is composed by A.

Subtyping

Often when modelling we need to specify classifications or categories of entities in our system. Eg There may exist 2 categories of employees, Junior and Senior. Employee is said to be the **supertype** while the subcategories Junior and Senior are referred to as **subtypes.** However there are different approaches to subtyping.

Enumeration

This is where a string column is simply used to indicate the subtype from a list of valid types



Use this when the type can be changed at runtime and there is no type specific behaviour that the class needs. Eg a method that only Junior employees have.

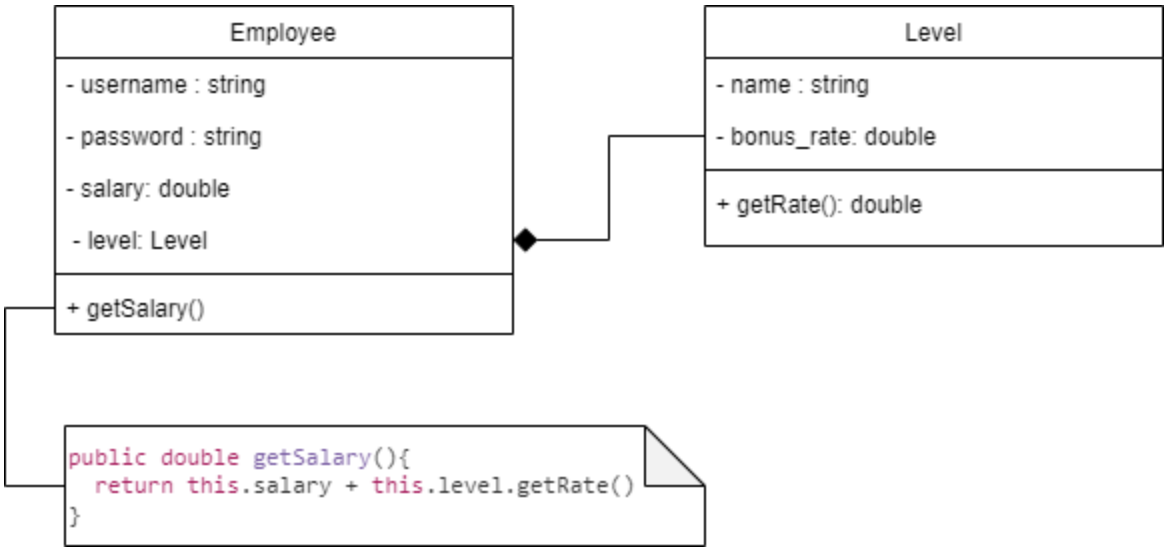
Composition

Here we dedicate a class to represent the concept of a level. Instances can then be created to represent the subtypes junior and level each with different states i.e. bonus rates.

An employee can then store the appropriate level object to calculate the salary.

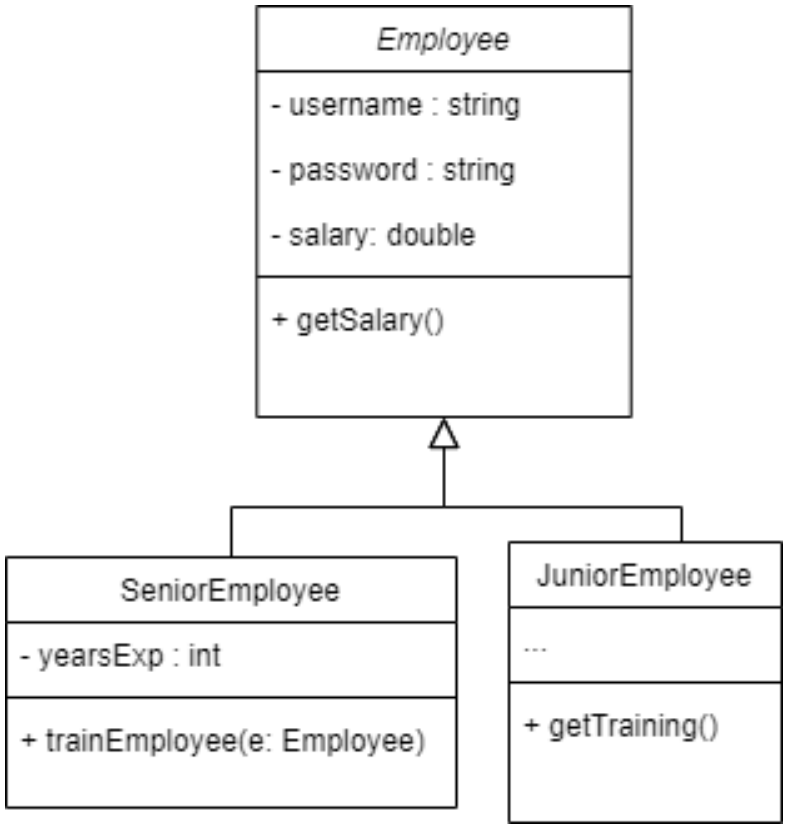
Use this when your subtype must include additional metadata and alters some behaviour of the composing class.

```
// the employee composes its level
Employee Bob = Employee(junior);
Employee Sally = Employee(senior);

// At runtime bob can be promoted
Bob.setLevel(senior)
```



Inheritance

In inheritance we make subclasses such the the supertype is the parent and the subtypes are children that inherit shared properties defined in the parent. Use this when your subtypes must add new state and/or behaviours to the parent type and when subtypes cannot be changed in runtime.



Note in this approach promoting an employee is not as straightforward. We must create replace an instance of one subtype with an instance of the other.

Model Specification with SQLAlchemy

As with any Object Relational Mapper, SQLAlchemy will create a table for every class in your design, Therefore you should be mindful of the complexity of your design. Additionally, not every piece of logic may need a class, **you can still write helper functions**.

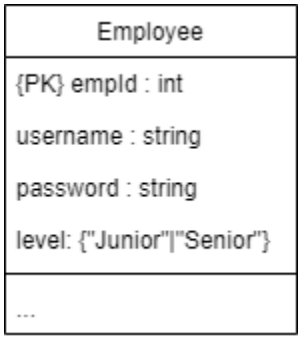
In this course when specifying SQLAlchemy models in our Model Design the following convection shall be adopted. We are specifying class diagrams but with the additional relational details we usually specify in Entity Relationship Diagrams ie PrimaryKeys and Foreign Keys

General guidelines

1. Identify all primary and foreign key fields
2. Foreign key fields should point to their corresponding primary key field with composition arrows 
3. Access specifiers (public, private) not necessary
4. Interfaces are not as useful non-typed python but abstract classes are
5. Multiple inheritance is allowed

Enumeration

Enums in sqlalchemy requires using the db.Enum field type and an enum class.You may want to consider using an enum instead of adding multiple classes if your subtypes don't have any unique state or behaviour.

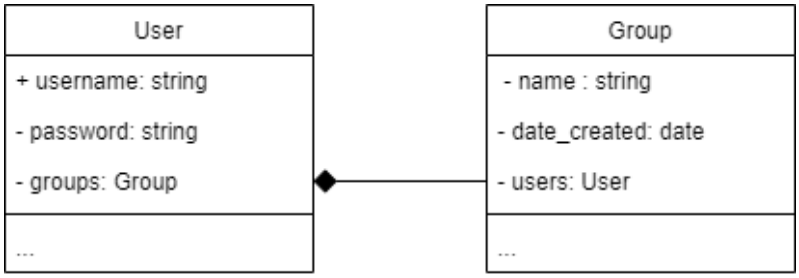



```python
import enum

class Level(enum.Enum):
    JUNIOR = "Junior"
    SENIOR = "Senior"

class Employee(db.Model):
  empId = db.Column(db.Integer, primary_key=True)
  username = db.Column(db.String(80), unique=True, nullable=False)
  password = db.Column(db.String(120), nullable=False)
  level = db.Column(db.Enum(Level), nullable=False, default=Level.JUNIOR)
```

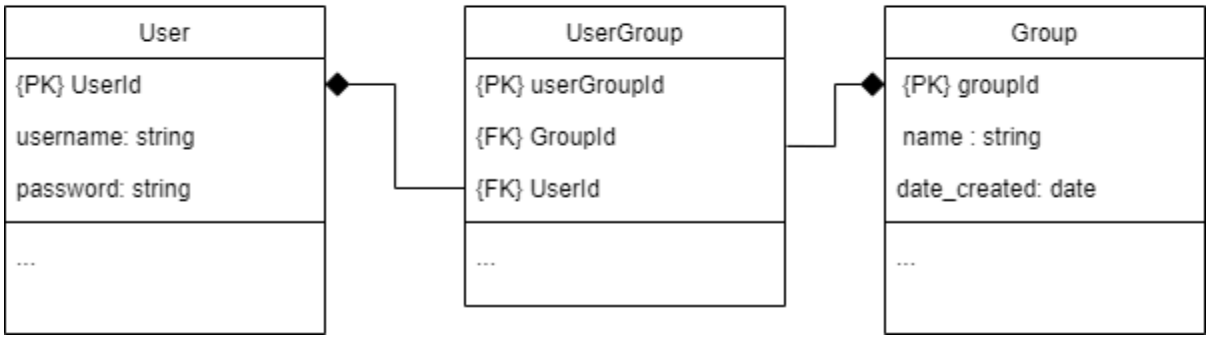
Composition

Composition is achieved in ERDs via relationships (1 to 1, 1 to Many & Many to Many).

For example the following design in java where a user can belong to multiple groups and a group can contain multiple users as follows:



Would become the following in SQLAlchemy introducing a bridge table to handle the many-to-many relationship on the database level.



Note how the foreign keys point to their primary keys using composition arrows.
The implementation is as follows

```
class User(db.Model):
  userId = db.Column(db.Integer, primary_key=True)
  username = db.Column(db.String(80), unique=True, nullable=False)
  password = db.Column(db.String(120), nullable=False)

class UserGroup(db.Model):
  userGroupId = db.Column(db.Integer, primary_key=True)
  userId = db.Column(db.Integer, db.ForeignKey('user.userId'), nullable=False)
  groupId = db.Column(db.Integer, db.ForeignKey('group.groupId'), nullable=False)

class Group(db.Model):
  groupId= db.Column(db.Integer, primary_key=True)
  name= db.Column(db.String(80), unique=True, nullable=False)
date_created= db.Column(db.Datetime)
```

Relationship fields are convenient fields you can put in your model so that you can access all related objects from another object. Eg we can get all the groups from User A via A.groups.

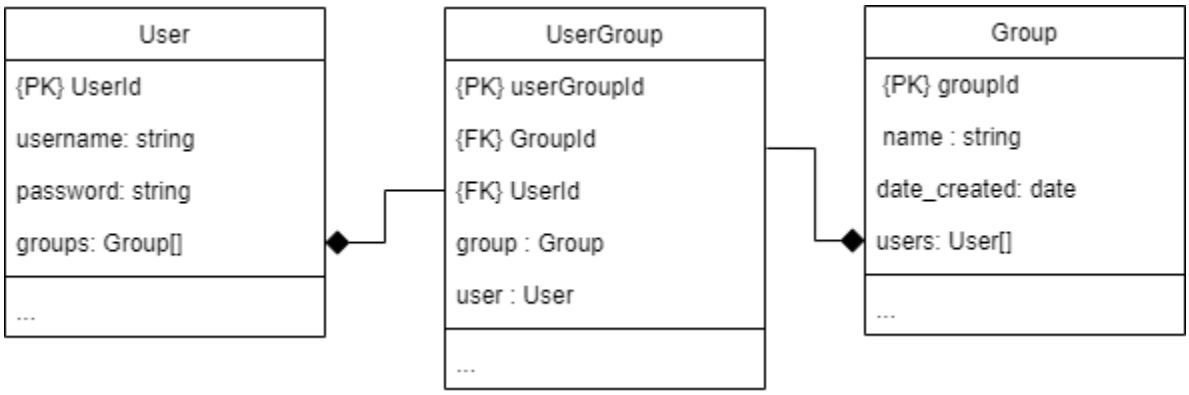
Relationship fields lets you do this

```
print(Bob.groups)
```

instead of

```
usergroups = UserGroup.query.filter_by(UserId=Bob.id).all() # get all records in bridge that belong to user
groups = [ Group.query.get(ug.groupID) for ug in usergroups  ] # get all the groups by id
print(groups)
```

If you opt to use relationship fields, include them in your diagram and specify the types accordingly. Note as this a many to many the edge table would contain collections ie Group[], User[].



The implementation is as follows:

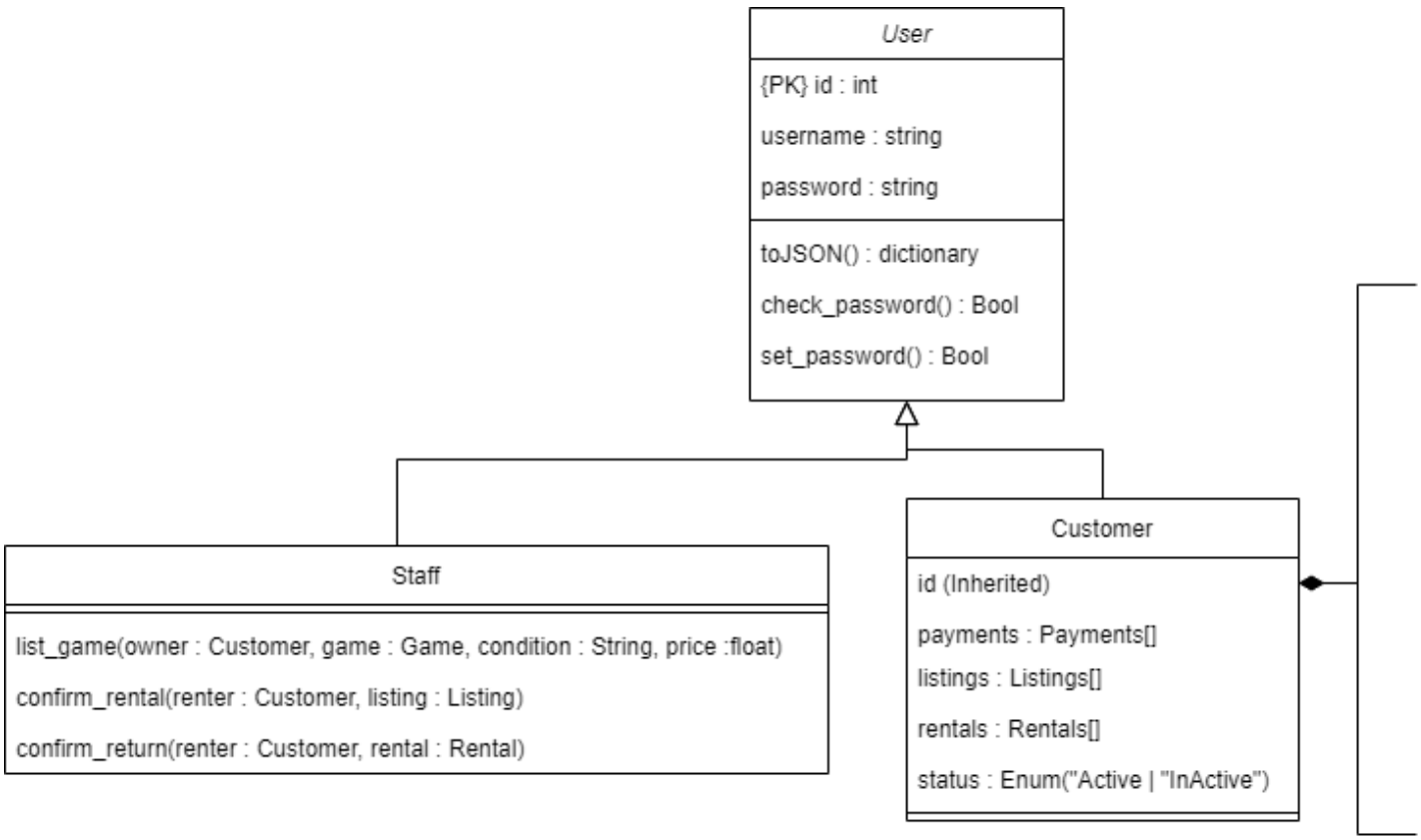
```
class User(db.Model):
  userId = db.Column(db.Integer, primary_key=True)
  username = db.Column(db.String(80), unique=True, nullable=False)
  password = db.Column(db.String(120), nullable=False)
  groups = db.relationship('UserGroup',  backref=db.backref('user', lazy='joined'))
#creates field user in UserGroup

class UserGroup(db.Model):
  userGroupId = db.Column(db.Integer, primary_key=True)
  userId = db.Column(db.Integer, db.ForeignKey('user.userId'), nullable=False)
  groupId = db.Column(db.Integer, db.ForeignKey('group.groupId'), nullable=False)
  # backref fields (group, user) are available here
```

```
class Group(db.Model):
  groupId= db.Column(db.Integer, primary_key=True)
  name= db.Column(db.String(80), unique=True, nullable=False)
  date_created= db.Column(db.Datetime)
  users = db.relationship('UserGroup',  backref=db.backref('group', lazy='joined'))
```

Note UserGroup.group and UserGroup.user are relationship fields with backreferences groups and users respectively.

Inheritance

In flask inheritance can be implemented in multiple ways (see inheritance link in references). Inheritance is very useful for defining user classes. The implementation here makes the parent class abstract and the children as separate concrete tables.



The child classes have common and specific state and behaviours.

References

1. UML Arrows Guide
2. SQLAlchemy Relationships
3. SQLAlchemy Inheritance