Composition over Inheritance & ORM

Dhatrika Nagalakshmi
Illinois Institute of Technology

ABSTRACT

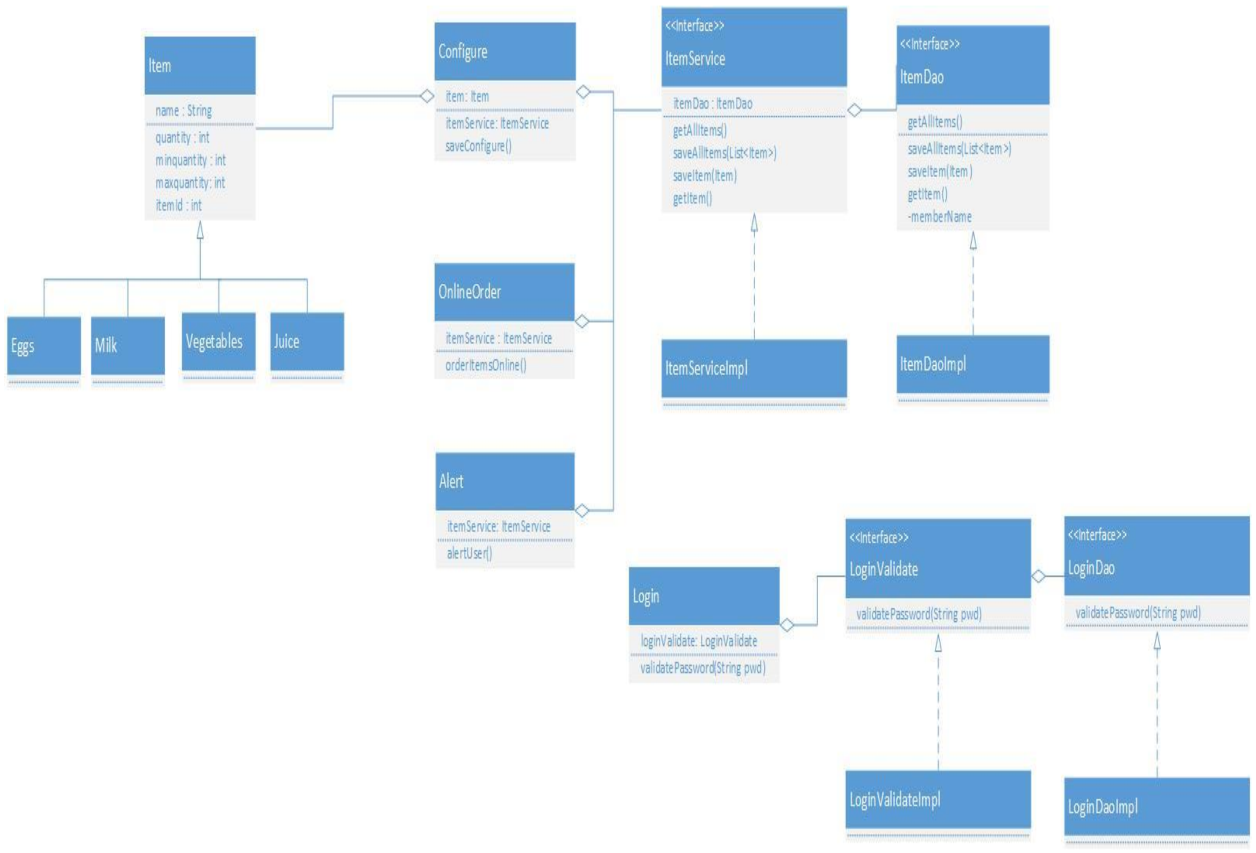
Composition over inheritance (or Composite Reuse Principle) in object-oriented programming is a technique by which classes may achieve polymorphic behavior and code reuse by containing other classes that implement the desired functionality instead of through inheritance.

INTRODUCTION

Inheritance is referred to as white-box reuse, with white-box referring to visibility, because the internals of parent classes are often visible to subclasses. In contrast, object composition in which objects with well-defined interfaces are used dynamically at runtime by objects obtaining references to other objects as black-box reuse because no internal details of composed objects need be visible in the code using them. An implementation of composition over inheritance typically begins with the creation of various interfaces representing the behaviors that the system must exhibit. The use of interfaces allows this technique to support the polymorphic behavior that is so valuable in object-oriented programming. Classes implementing the identified interfaces are built and added to business-domain classes as needed. Thus, system behaviors are realized without inheritance. In fact, business-domain classes may all be base classes without any inheritance at all. Alternative implementation of system behaviors is accomplished by providing another class that implements the desired behavior interface. Any business-domain class that contains a reference to the interface can easily support any implementation of that interface and the choice can even be delayed until run time.

MVC (Model View Controller) & Three-tiered applications

The main usage of this principle is the today's MVC (Model View Controller) model which is used across most of three -tiered web applications. Three-tiered applications have three layers: Dao (database), Service (Business) and Web (Action). Dao (Data Access Object) interface objects are used as instance variables in the Service layer (business layer) implementation classes. Dao instance variables can be initialized using "Spring" framework's dependency injection. Similarly, business/service layer interface objects are used as instance variables in the Web layer implementation classes (action classes). The primary advantage behind this approach is the independence of each layer from the layer beneath it. For instance, if an in-house web application of a company/firm had been using MS SQL initially and wants to switch/upgrade to a completely new database system (for ex: Oracle) then it's only the Dao layer's implementation classes that needs to be changed without majorly changing the layers above it. Business/Service layer and Web/Action layer classes will remain intact.



Above shows class diagram of a Smartfridge application which is a web application that has three tiers as mentioned above. It shows that ItemDao is composed in ItemService. ItemService is in turn composed in the Configure (Action/Web) class. So, if there was a situation when database had to be changed from MS SQL to Oracle, then ItemDaoImpl and LoginDaoImpl are the only classes which will have to be modified. Usage of interfaces and composition go hand in hand which is the main reason why we are able to accomplish the MVC model for a web application.

CODE REUSE

Code reuse via inheritance: For an illustration of how inheritance compares to composition in the code reuse department, consider this very simple code snippet:

```
Class Fruit {
        // Returns int number of pieces of peel that
        // resulted from the peeling activity.
        public int peel() {
                System.out.println("Peeling is appealing.");
                return 1;
        }
}

Class Apple extends Fruit {
}

Class Example1 {
        public static void main(String[] args)
        {
                Apple apple = new Apple();
                int pieces = apple.peel();
        }
}
```

When you run the Example1 application, it will print out "Peeling is appealing.", because Apple inherits (reuses) Fruit's implementation of peel(). If at some point in the future, however, if we wish to change the return value of peel() to object type “Peel”, it would break the code for Example1. Our notion to change the Fruit breaks Example1's code even though Example1 uses Apple directly and never explicitly mentions Fruit.

Here's what above would look like:

```
Class Peel
{
            private int peelCount;
            public Peel(int peelCount)
            {
                        this.peelCount = peelCount;
            }
            public int getPeelCount() {
                        return peelCount;
            }
//...
}

Class Fruit {
            // Return a Peel object that
            // results from the peeling activity.
            public Peel peel() {
                        System.out.println("Peeling is appealing.");
                        return new Peel(1);
                        }
}

// Apple still compiles and works fine
Class Apple extends Fruit {
 }

// this old implementation of Example1 // is broken and won't compile.
Class Example1 {
            public static void main(String[] args) {
                        Apple apple = new Apple();
                        int pieces = apple.peel();
            }
}
```

Code reuse via composition: Composition provides an alternative way for Apple to reuse Fruit's implementation of peel(). Instead of extending Fruit, Apple can hold a reference to a Fruit instance and define its own peel() method that simply invokes peel() on the Fruit.

Here's the code:

```
Class Fruit {
        // Return int number of pieces of peel that // resulted from the peeling
        activity.
        public int peel() {
                System.out.println("Peeling is appealing.");
                return 1;
        }
}

Class Apple {
        private Fruit fruit = new Fruit();
        public int peel() {
                return fruit.peel();
        }
}

class Example2 {
        public static void main(String[] args) {
                Apple apple = new Apple();
                int pieces = apple.peel();
        }
}
```

The composition approach to code reuse provides stronger encapsulation than inheritance, because a change to a back-end class (i.e. Fruit) needn't break any code that relies only on the front-end class (i.e. Apple). For example, changing the return type of Fruit's peel() method from the previous example doesn't force a change in Apple's interface and therefore needn't break Example2's code.

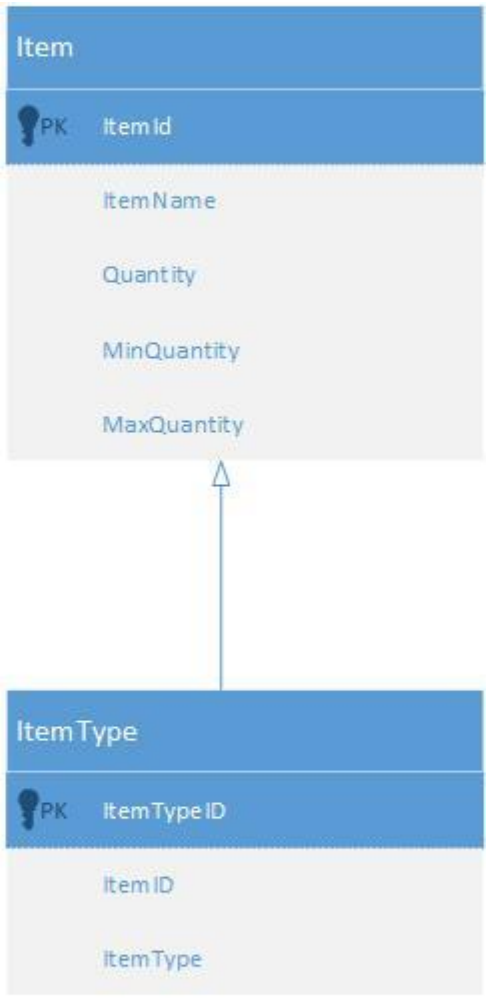
Here's how the changed code would look:

```
Class Peel {
        private int peelCount;
        public Peel(int peelCount) {
                this.peelCount = peelCount;
         }
        public int getPeelCount()
        {return peelCount;
        }
//...
}

Class Fruit {
        // Return int number of pieces of peel that
        // resulted from the peeling activity.
        public Peel peel()
        {
                System.out.println("Peeling is appealing.");
                return new Peel(1);
        }
}

// Apple must be changed to accommodate the change to Fruit

Class Apple {
        private Fruit fruit = new Fruit();
        public int peel() {
                Peel peel = fruit.peel();
                return peel.getPeelCount();
        }
}

// this old implementation of Example2
// still works fine.
Class Example1 {
        public static void main(String[] args) {
                Apple apple = new Apple();
                int pieces = apple.peel();
        }
}
```

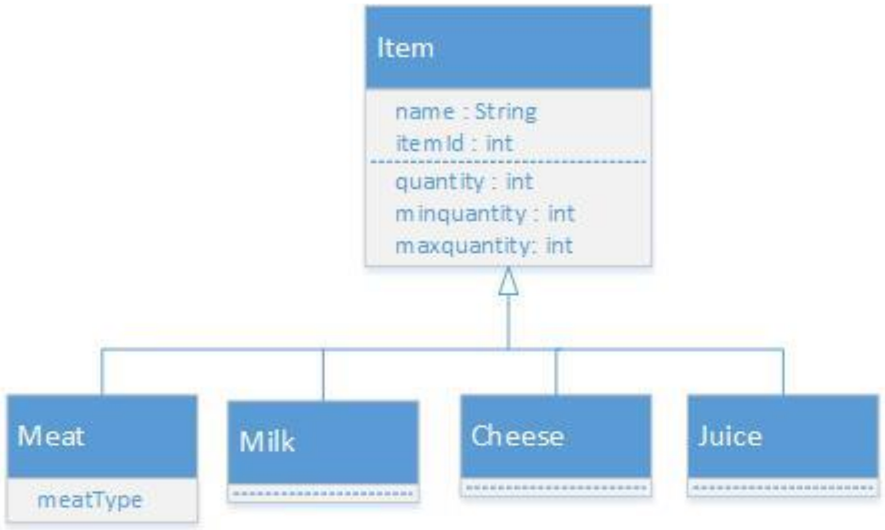
Inheritance using ORM tools

ORM tools such as hibernate and ibatis support inheritance without any complexity. Also, it’s not complicated to model inheritance in SQL. It can mostly be achieved in similar lines as Composition (i.e. using foreign - key relationship) but rather in a different way.

For ex: Suppose there is a table/entity which represents an Item in a Smartfridge application. An Item has got ItemId, ItemName, Quantity, MaxQuantity and MinQuantity. This is general info (generalization) which is common to all the Items. Suppose there is a specific item like "Meat" which has a "Type" and is not applicable for all the other items like Milk, Juice and Cheese then our database tables can be modeled as:



We can model the above in java as:



Above shows that “Meat” is the only Item which has the specific information i.e. meatType

Below illustrates how inheritance modelled in SQL can be used /mapped to Java model using JDBC (java mapping to database) and other ORM (Object Relational Mapping) tools.

1. MODELLING USING JDBC:

Item “Meat” can be defined and initialized from the database using the following pseudo code using JDBC / ODBC.

```
Prepared database Statement:
        Select itm.ItemId, itm.ItemName, itm.Quantity, itm.MinQuantity,
        itm.MaxQuantity, itm.Type
        from Item itm
        JOIN ItemType itmType on (item.ItemId = itmType.ItemID)
        Where itm.ItemID = <meatID> OR item.ItemName = "Meat"

ResultSet rs = preparedStatement.execute();

while(rs.next()){
        Meat meat = new Meat;
        meat.setItemId(rs.getInt(1));
        meat.setItemName(rs.getString(2));
        meat.setMinQuantity(rs.getInt(3));
        meat.setMaxQuantity(rs.getInt(4));
        meat.setItemType(rs.getString(5));
}
```

2. MODELLING USING HIBERNATE:

```
<class name=Item" table="Item">
   <id name="id" type="long" column="ItemID">
      <generator class="native"/>
   </id>
   <property name="itemName" column="ItemName"/>
   <property name="quantity" column="Quantity"/>
   <property name="minQUantity" column="MinQuantity"/>
   <property name="maxQuantity" column="MaxQuantity"/>

   <joined-subclass name="Meat" table="ItemType">
      <key column="ItemID"/>
      <property name="meatType" column="itemType"/>
   </joined-subclass>
</class>
```

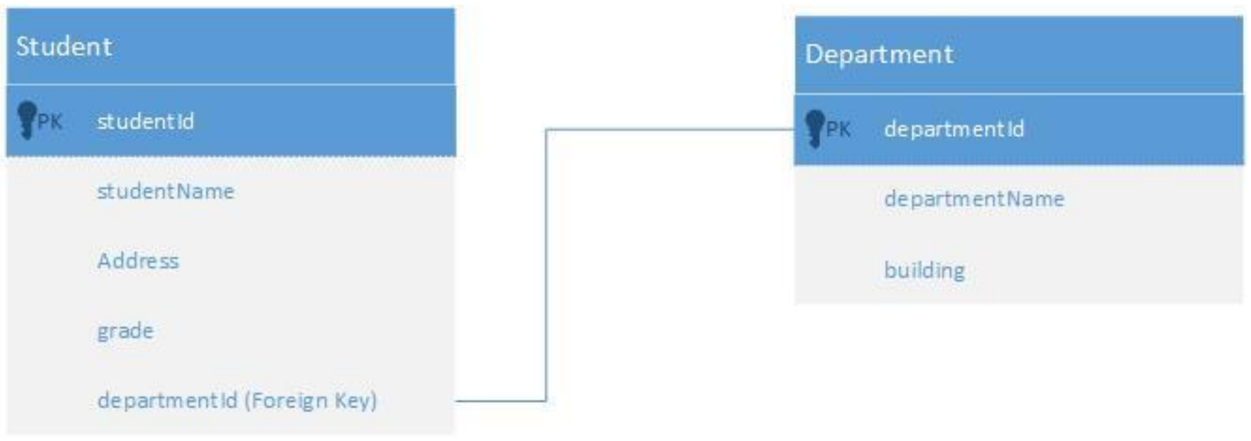
3. MODELLING USING IBATIS:

```
<resultMaps>
  <resultMap id="select-item-result" class="Item">
            <property="itemID" column="ItemID"/>
            <property="itemName" column="ItemName"/>
            <property="quantity" column="Quantity" />
            <property="minQuantity" column="MinQuantity" />
            <property="maxQuantity" column="MaxQuantity" />
  </resultMap>
  <resultMap id="select-meat-result" class="Meat" extends="select-item-result">
            <property="meatType" column="itemType"/>
  </resultMap>
</resultMaps>

<statements>
  <select id="selectItem" parameterClass="int" resultMap="select-item-result">
   select * from Item where ItemID = #value#
  </select>

  <select id="selectMeat" parameterClass="int" resultMap="select-meat-result">
   select * from Item itm
   JOIN ItemType itmType on (item.ItemId = itmType.ItemID)
   Where itm.ItemID = #value#
  </select>
</statements>
```

Composition using ORM tools

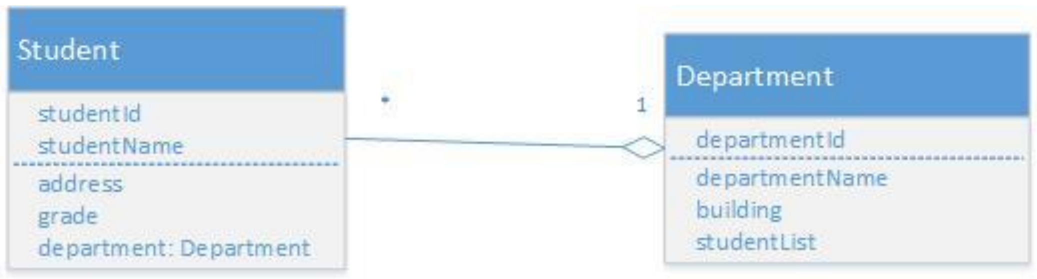
ORM tools such as hibernate and ibatis support composition. It can be achieved using foreign key relationship.

For ex: Suppose there is a table/entity which represents a student in the entity department. Student belongs to a department. DepartmentID can be specified as the foreign key in the student table. These entities can be modelled using database tables as below:



The same entities can be modelled in java as:

A department can have more than one student. A student belongs to a department.



1. MODELLING USING HIBERNATE:

```
<class name=Student" table="Student">
  <id name="id" type="long" column="StudentId">
    <generator class="native"/>
  </id>
  <property name="studentName" column="StudentName"/>
  <property name="address" column="Address"/>
  <join table="Department">
    <key column="departmentId"/>
    <property name="departmentName" column="departmentName"/>
    <property name="building" column="building"/>
  </join>
</class>
```

2. MODELLING USING IBATIS:

```
<resultMaps>
 <resultMap id="select-student-result" class="Student">
        <property="studentId" column="studentID"/>
        <property="studentName" column="studentName"/>
        <property="address" column="Address" />
        <property="department.departmentId" column="departmentId" />
        <property="department.departName" column="departmentName" />
        <property="department.building" column="building" />
 </resultMap>
</resultMaps>

<statements>
 <select id="selectStudent" parameterClass="int" resultMap="select- student -result">
        select  st.studentId, st.studentName, st.address, st.departmentId,
        dt.departmentName,  dt.building
        from Student st
        JOIN department dt on (dt.departmentId = st.departmentID)
         where st.studentId = #value#
 </select>

</statements>
```

Conclusion

Composition and Inheritance both can be achieved and modelled using the ORM tools (both in the database as well as its mapping to the programming language / business entities). Inheritance can be used to model generalization and specialization among the business entities. However, Composition is preferred over Inheritance due to its innate nature of being “loosely coupled”. Both Inheritance and Composition can be used for modelling the business entities depending upon the use cases and nature of relationship between the business entities.

References

1. http://en.wikipedia.org/wiki/Composition_over_inheritance
2. http://en.wikipedia.org/wiki/Design_Patterns_%28book%29
3. http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance
4. http://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html
5. http://docs.jboss.org/hibernate/orm/3.5/reference/en-US/html/inheritance.html
6. https://ibatis.apache.org/docs/dotnet/datamapper/ch03s05.html
7. http://stackoverflow.com/questions/13316472/uml-composition-vs-sql-foreign-key