Object-Oriented Principals

Encapsulation and Delegation

Good Software

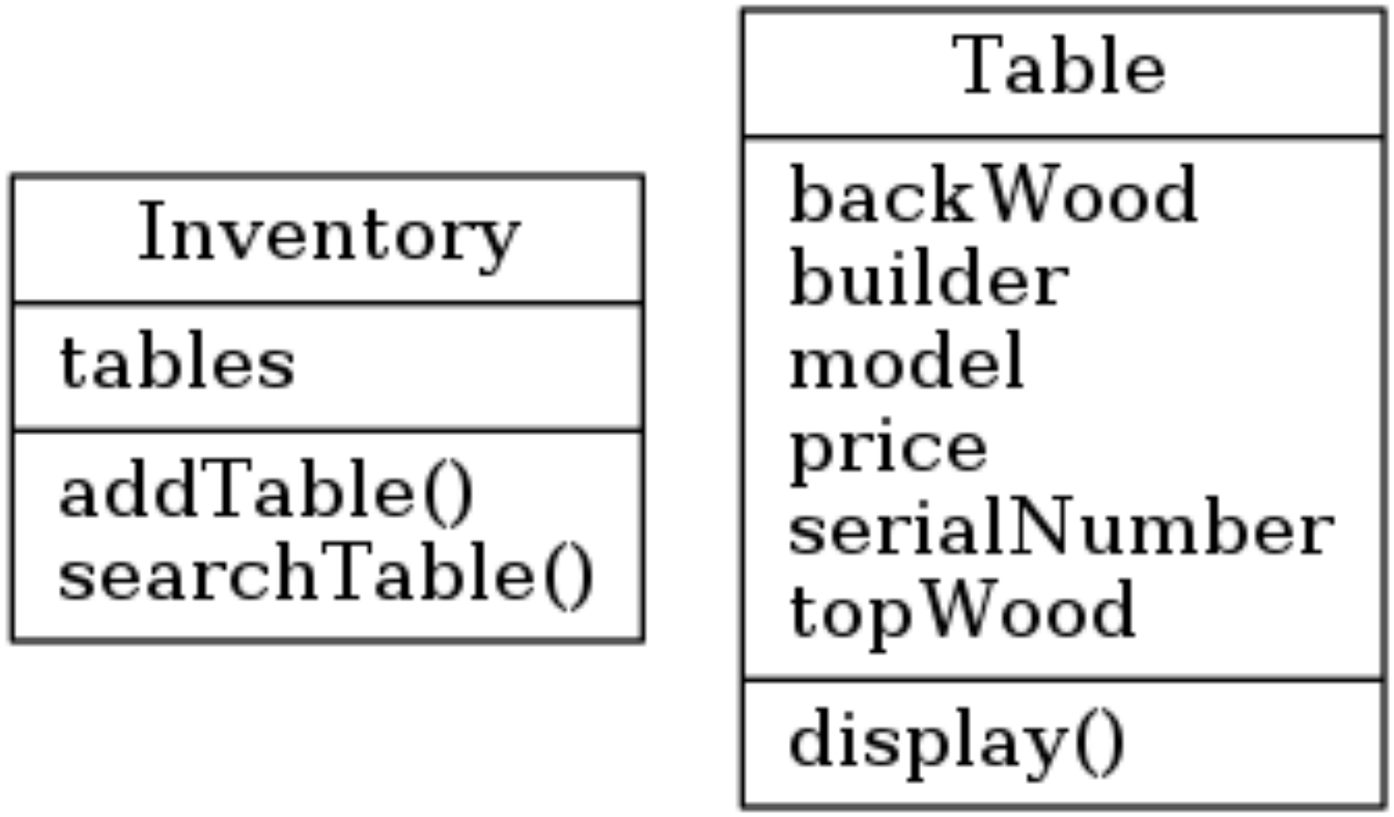
1. Make sure your software meets or exceeds the **customer's requirements**. Even though you're working on **functionality** in this step, you still can use OO principles to make sure your software is well designed from the beginning.

2. Apply **object-oriented principle**s to reduce duplicated code and add flexibility.

3. Make sure your software has a **maintainable, reusable design**. So, it can be used for years.

Case Study - Table Inventory System

- The table inventory system has two classes as shown below.
- There is a problem in the search table method, it does not search the table consistently based on the table specification.



The existing implementation of the searchTable() method

- Can you find out the problem? Or can you find anything to improve?

```
def searchTable(self, targetTable):
        aTable = None
        for table in self.__tables:
            if table.builder != targetTable.builder:
                continue
            if table.model != targetTable.model:
                continue
            if table.backWood != targetTable.backWood:
                continue
            if table.topWood != targetTable.topWood:
                continue
            aTable = table
            break
        return aTable
```

Tester

```
def main():
    inventory = Inventory()
    inventory.addTable("123456", 1000, "builderA", "modelA", "woodA", "woodA")
    inventory.addTable("223456", 1100, "builderA", "modelA", "woodA", "woodA")
    inventory.addTable("323456", 1200, "builderB", "modelB", "woodA", "woodA")
    inventory.addTable("423456", 1300, "builderA", "modelA", "woodC", "woodD")
    inventory.addTable("523456", 1400, "builderA", "modelA", "woodC", "woodD")

    targetTable = Table("", 0, "builderA", "modelA", "woodA", "woodA")
    table = inventory.searchTable(targetTable)
    if table is not None:
        table.display()
```

Using Enum

- Though Enum is not the OOP principal, it can help to make your software to achieve the expected functionality
- You should use Enum to avoid string comparisons. The string comparison may cause errors that could be caused by mismatched cases and spellings.

```
import enum

class Builders(enum.Enum):
        builderA = 1
        builderB = 2
        builderC = 3

class Models(enum.Enum):
        modelA = 1
        modelB = 2
        modelC = 3

class Wood(enum.Enum):
        woodA = 1
        woodB = 2
        woodC = 3


class Table:
        def __init__(self, serialNumber, price, builder, model, backWood, topWood):
                self.__serialNumber = serialNumber
                self.__price = price
                self.__builder = builder
                self.__model = model
                self.__backWood = backWood
                self.__topWood = topWood
```

Tester

```
def main():
        inventory = Inventory()
        inventory.addTable("123456", 1000, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        inventory.addTable("223456", 1100, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        inventory.addTable("323456", 1200, Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC)
        inventory.addTable("423456", 1300, Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC)
        inventory.addTable("523456", 1400, Builders.builderC, Models.modelC, Wood.woodA, Wood.woodC)

        targetTable = Table("", 0, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        table = inventory.searchTable(targetTable)
        if table is not None:
                table.display()
```

Mismatched Object Type

- Objects should do what their names indicate.
- Each object should represent a single concept.
  - Each object should not serve double or triple duty.
- Unused properties should be removed.
  - If you've got an object that is being used with no-value or null properties often, you've probably got an object doing more than one job.
  - Let's look at the following code, you have to create a table object in order to pass the table specification to the searchTable() to search the tables you want. Notice that the serial number and price are not used here.

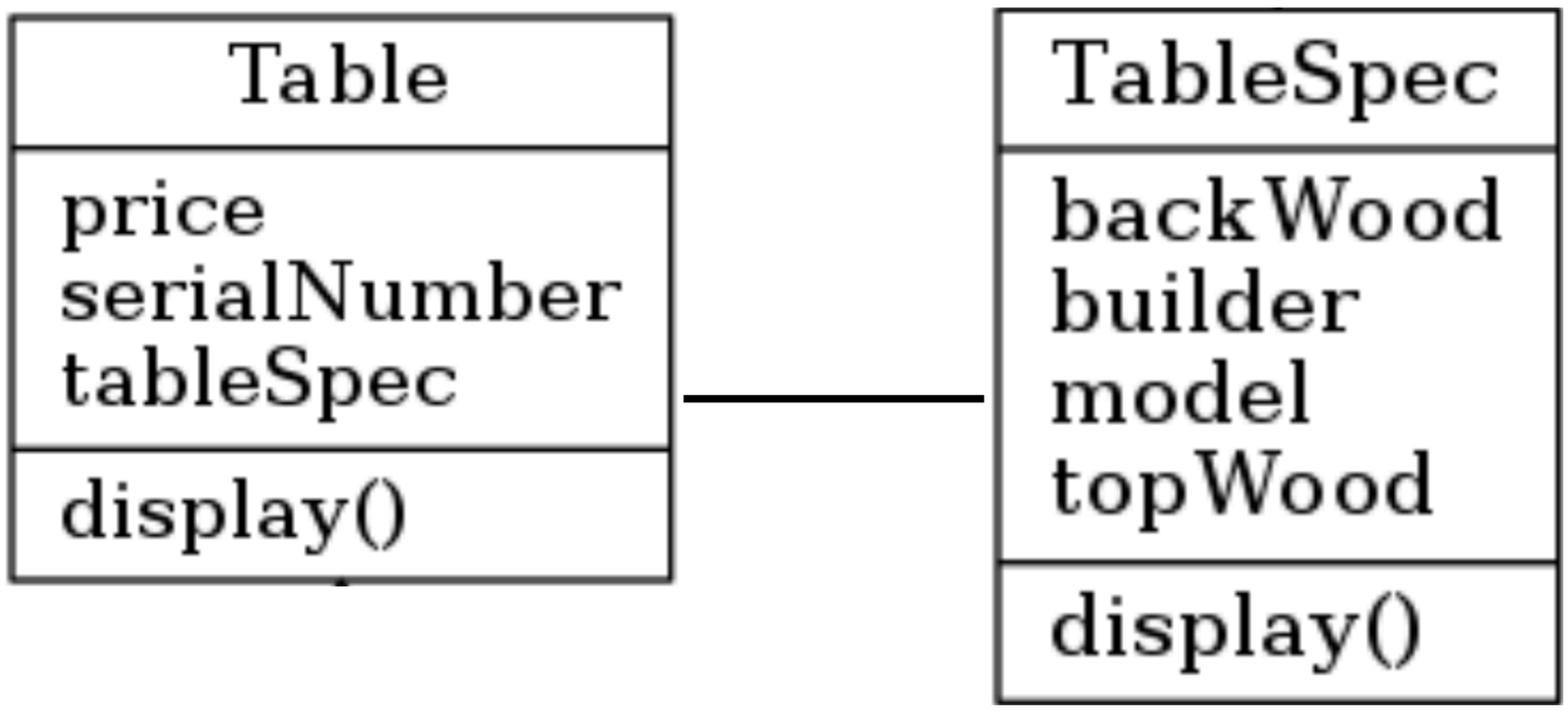
```
targetTable = Table("", 0, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
table = inventory.searchTable(targetTable)
```

Encapsulation

- The idea behind encapsulation is to protect the data in your class from the rest of your app by making that data private. But sometimes the information might be an entire set of attributes like the details about a table or even behavior like how a person performs his job.
- When you break that behavior out from a class, you can change the behavior without the class having to change as well. So if you changed how attributes were stored, you wouldn’t have to change your Table class at all, because the attributes are encapsulated away from Table.
- That’s the power of encapsulation: by breaking up the different parts of your app, you can change one part without having to change all the other parts. In general, you should encapsulate the parts of your app that might vary away from the parts that will stay the same.

Encapsulation

- We encapsulated those attributes related to table specification out from the original Table class. We left serialNumber and price left in the Table class.



Encapsulation

```
class TableSpec:
        def __init__(self, builder, model, backWood, topWood):
                self.__builder = builder
                self.__model = model
                self.__backWood = backWood
                self.__topWood = topWood

        @property
        def builder(self):
                return self.__builder

        @property
        def model(self):
                return self.__model

        @property
        def backWood(self):
                return self.__backWood

        @property
        def topWood(self):
                return self.__topWood

        def display(self):
                print("builder = ", self.builder.name)
                print("model = ", self.model.name)
                print("topWood = ", self.topWood.name)
                print("backWood = ", self.backWood.name)
```

```
class Table:
        def __init__(self, serialNumber, price, builder, model,
backWood, topWood):
                self.__serialNumber = serialNumber
                self.__price = price
                self.__tableSpec = TableSpec(builder, model,
backWood, topWood)

        @property
        def serialNumber(self):
                return self.__serialNumber

        @property
        def price(self):
                return self.__price

        @price.setter
        def price(self, price):
                self.__price

        @property
        def tableSpec(self):
                return self.__tableSpec

        def display(self):
                print("serialNumber = ", self.serialNumber)
                print("price = ", self.price)
                self.__tableSpec.display()
```

Encapsulation

```
class Inventory:
        def __init__(self):
                self.__tables = []

        @property
        def tables(self):
                return self.__tables

        def addTable(self, serialNumber, price, builder, model, backWood, topWood):
                table = Table(serialNumber, price, builder, model, backWood, topWood)
                self.__tables.append(table)

        def searchTable(self, tableSpec):
                matchedTables = []
                for table in self.__tables:
                        if table.tableSpec.builder != tableSpec.builder:
                                continue
                        if table.tableSpec.model != tableSpec.model:
                                continue
                        if table.tableSpec.backWood != tableSpec.backWood:
                                continue
                        if table.tableSpec.topWood != tableSpec.topWood:
                                continue
                        matchedTables.append(table)
                return matchedTables
```

Instead of creating a table object, now we can create a table spec object

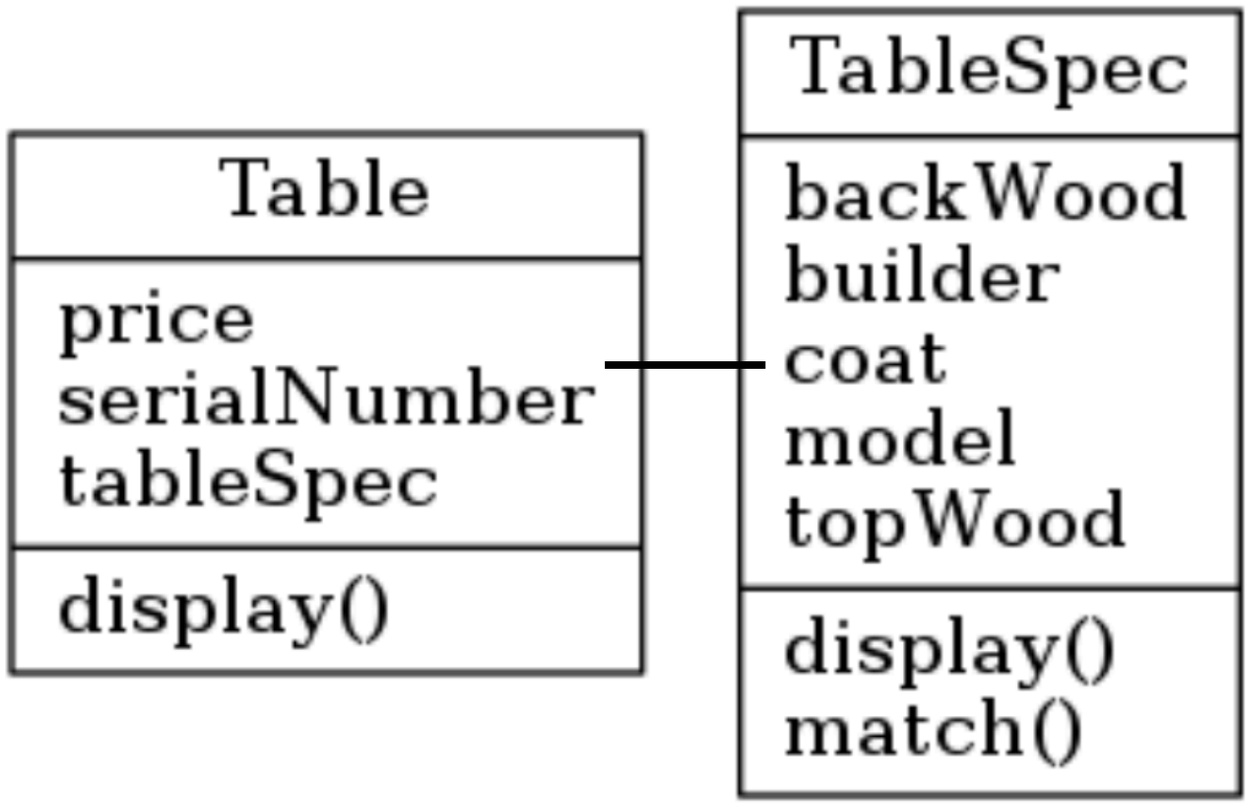
```
def main():
        inventory = Inventory()
        inventory.addTable("123456", 1000, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        inventory.addTable("223456", 1100, Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        inventory.addTable("323456", 1200, Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC)
        inventory.addTable("423456", 1300, Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC)
        inventory.addTable("523456", 1400, Builders.builderC, Models.modelC, Wood.woodA, Wood.woodC)

        tableSpec = TableSpec(Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA)
        matchedTables = inventory.searchTable(tableSpec)
        for table in matchedTables:
                table.display()
                print()
```

Delegation

- Delegation is when an object needs to perform a certain task, and instead of doing that task directly, it asks another object to handle the task.
- Delegation makes your code more reusable. It also lets each object worry about its own functionality.
- Delegation makes your objects are more independent of each other, or more loosely coupled. Loosely coupled objects can be taken from one app and easily reused in another.
- Loosely coupled applications are usually more flexible, and easy to change. Since each object is pretty independent of the other objects.

Delegation for Maintainability

- Let say we need to add a new attribute to the table specification to define a coating for the table. We would need to modify the TableSpec class. However, we will also need to modify the Inventory class as well as Table class with the current implementation.
- The below is the new class diagram after the new attribute “coat:” was added.



Delegation

We no longer need to change the Table class in case we need to add a new attribute in the TableSpec class.

```
class TableSpec:
        def __init__(self, builder, model, backWood, topWood, coat):
                self.__builder = builder
                self.__model = model
                self.__backWood = backWood
                self.__topWood = topWood
                self.__coat = coat

        @property
        def builder(self):
                return self.__builder

        @property
        def model(self):
                return self.__model

        @property
        def backWood(self):
                return self.__backWood

        @property
        def topWood(self):
                return self.__topWood

        @property
        def coat(self):
                return self.__coat

        def display(self):
                print("builder = ", self.builder.name)
                print("model = ", self.model.name)
                print("topWood = ", self.topWood.name)
                print("backWood = ", self.backWood.name)
                print("coat = ", self.coat.name)

        def match(self, tableSpec):
                if self.__builder != tableSpec.builder:
                        return False
                if self.__model != tableSpec.model:
                        return False
                if self.__backWood != tableSpec.backWood:
                        return False
                if self.__topWood != tableSpec.topWood:
                        return False
                if self.__coat != tableSpec.coat:
                        return False
                return True
```

```
class Table:
        def __init__(self, serialNumber, price, tableSpec):
                self.__serialNumber = serialNumber
                self.__price = price
                self.__tableSpec = tableSpec

        @property
        def serialNumber(self):
                return self.__serialNumber

        @property
        def price(self):
                return self.__price

        @price.setter
        def price(self, price):
                self.__price

        @property
        def tableSpec(self):
                return self.__tableSpec

        def display(self):
                print("serialNumber = ", self.serialNumber)
                print("price = ", self.price)
                self.__tableSpec.display()
```

Delegation

```
class Inventory:
        def __init__(self):
                self.__tables = []

        @property
        def tables(self):
                return self.__tables

        def addTable(self, serialNumber, price, tableSpec):
                table = Table(serialNumber, price, tableSpec)
                self.__tables.append(table)

        def searchTable(self, tableSpec):
                matchedTables = []
                for table in self.__tables:
                        if not table.tableSpec.match(tableSpec):
                                continue
                        matchedTables.append(table)
                return matchedTables
```

We no longer to need to change the searchTable() method in case there is a change in TableSpec class because we delegated the match() method to the TableSpec class

```
def main():
        inventory = Inventory()
        inventory.addTable("123456", 1000, TableSpec(Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA, Coat.coatA))
        inventory.addTable("223456", 1100, TableSpec(Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA, Coat.coatA))
        inventory.addTable("323456", 1200, TableSpec(Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC, Coat.coatC))
        inventory.addTable("423456", 1300, TableSpec(Builders.builderB, Models.modelB, Wood.woodB, Wood.woodC, Coat.coatB))
        inventory.addTable("523456", 1400, TableSpec(Builders.builderC, Models.modelC, Wood.woodA, Wood.woodC, Coat.coatB))

        tableSpec = TableSpec(Builders.builderA, Models.modelA, Wood.woodA, Wood.woodA, Coat.coatA)
        matchedTables = inventory.searchTable(tableSpec)
        for table in matchedTables:
                table.display()
                print()
```

Summary

- Functionality - make your software satisfy the customer by doing what the customer wants it to do.
- Encapsulation - keep the parts of your code that stay the same separate from the parts that change; then your software is more flexible so it’s really easy to make changes to your code without breaking everything.
    - Find the parts of your application that change often, and try and separate them from the parts of your application that don't change.
- Delegation - is giving another object the responsibility of handling a particular task. Therefore, your code becomes more reusable.