Sean Livingston
SP21:Objected Oriented Design & Program
March 8th, 2021
Assignment 1

1. Information hiding in software design is the organizing of objects in such a manner that only certain attributes and methods are publicly available while hiding many parts of an object that do not need to be exposed. For example: If a Television represents the whole of a piece of software, only portions of it need to be exposed to the user. The user needs to know how to operate the television in a few basic ways. Some of these ways include: knowing the volume number, the channel number, changing the channel, changing the volume, and powering the television on and off. These are the parts of the television that need to be exposed to the user. How the television actually performs these actions internally is not something that is necessary for the user to know. If the user has access to the inner workings of the television, it is very likely that they may change something that can break the software entirely.

2. Class-Responsibility-Collaboration cards are part of an exercise to help design and organize software. First one must write down the class names on index cards. Next to each class, one should list the class's responsibilities and the classes (Collaborators) that will collaborate with said class. Responsibilities to be listed include short sentences or phrases that describe the actions the object should perform, the knowledge the object maintains, as well as some important decisions the object makes. The collaborators that are to be listed are other classes that must work with said class. If other classes will need messages sent by said class, then those classes should be listed as collaborators.

3. The SOLID design principle being violated is the "Single Responsibility Principle." The *Transportation* class is handling too many responsibilities. For example: transportation devices do not all contain engines; therefore the attribute *engine*, and methods getEngine(), setEngine(Engine e), and startEngine() should be extracted out of *TransportationDevice*, and only used in objects that extend *TransportationDevice* that require an engine to function, such as class *Car*. Attributes *name and speed*, as well as methods setName(String n), getSpeed(), getName(), and setSpeed(double d) should all remain within the *TransportationDevice* class. Class *Bicycle* has absolutely no need for the startEngine() method, so it should be removed entirely from this class. Both classes *Car* and *Bicycle* should include their own implementations of setSpeed() because a car and a bicycle accelerate in different ways and at different rates.

   ```
   class TransportationDevice
   {
   ```

```
  String name;
  String getName() {...}
  void setName(String n) {...}
  double speed;
  double getSpeed() {...}
  void setSpeed(double d) {...}
}

class Car extends TransportationDevice
{
  @Override
  void setSpeed(double d) {...}
  void setEngine(Engine e) {...}
  void startEngine() {...}
}

Class Bicycle extends TransportationDevice
{
  @Override
  void setSpeed(double d) {...}
  }
```

4. The OO SOLID design principle being violated is that of Interface Segregation. IEmployee is a fat interface in this design because work() and eat() will probably differ based on the type of employee. eat() also is not required of an employee; lunch is required of an employee, so this function should be refactored and given a different name, such as takingLunch(Time time, bool isTaken). takingLunch() should be placed in its own interface as *ILunch,* so that if robot employees are brought on, IEmployee does not contain requirements that are unneeded for a robot. I would then add a *LunchImpl* class that extends *iLunch* and add it as a property to the *Grade1Employee* and *Grade2Employee* classes. All robots should be instances of a new class called *RobotEmployee*, which implements the newly refactored *IEmployee*.

```
interface IEmployee {
  public void work();
}

interface ILunch {
  public bool takingLunch(Time time, bool isTaken);
}

class RobotEmployee implements IEmployee {
```

```
  public void work() {
    // working
  }
}

class Grade1Employee implements IEmployee{
  Lunch lunch = new LunchImpl();
  public void work()
  {
    //.... working more
  }
}

class Grade2Employee implements IEmployee{
  Lunch lunch = new LunchImpl();
  public void work()
  {
    //.... working more
  }
}

class LunchImpl implements iLunch {
  public bool takingLunch(Time time, bool isTaken) {
    // lunch code here
  }
}
```