M4 README Responses

Targeted Difficulty: Hard

Easy: Design Principle

1. We selected the principle of Favoring Composition Over Inheritance**.**
2. To follow this principle, it means that a class extends and inherits functionality from a class instead of composing an instance of that class as a field in order to obtain the same functionality.
3. A violation of our principle occurs when…
   1. Inheritance is not being used for the purposes of the Decorator pattern
   2. The class being inherited from does not have all abstract methods
4. Our project follows this principle perfectly, as we have checks for…
   1. Whether or not the super class has all abstract methods
   2. When the Decorator pattern is present

Our project violates this principle by having classes extend off of our abstract *DiagramElement* class. The reason we used inheritance for this was to be able to pass around the abstract *DiagramElement* type for ease of use when generating the Diagram which calls the *generate* method on all Diagram Elements.

1. The demo we selected is a run configuration comprised of six different cases of inheritance. In these six examples both non-violation and violation examples are given.

Medium: Decorator

* 1. The essential syntactic qualities of Decorator are…
     1. A class has a field of the same type of one of its super classes / interfaces OR a class inherits a field from one of its super classes of an object it decorates
     2. The Decorator has **ALL** the same methods (same name, same parameters, same return type) of the class / interface that is being decorated (the Component)
  2. In our code we determine something is a Decorator by…
     1. Iterate over each ClassElement (only classes can be Decorators)
     2. Obtain all of each class’ super classes and interfaces
     3. For each field…
        1. Check if the field matches one of the class’s super classes / interfaces
           1. Check to see if the field has a ClassElement or InterfaceElement (the field is not a primitive)
           2. If the field has a ClassElement or InterfaceElement check to see if the current class implements ALL of the methods of the ClassElement or InterfaceElement
           3. If this is the case mark the class and ClassElement/Interface Element as a decorator and component respectively
        2. For each of the current methods super classes…
           1. For each field of the current super class…

Check to see if the field has a ClassElement or InterfaceElement (the field is not a primitive)

If the field has a ClassElement or InterfaceElement determine if the field is accessible (public or protected) by the current class. This is important because the class must have access to the field in order to decorate something implements ALL of the methods of the ClassElement or InterfaceElement

Ensure that the field is not of the same type of the current super class (this could present a situation where a LinkedList where a Node has a next node; we want to avoid calling this a Decorator)

If this is the case mark the class and ClassElement/InterfaceElement as a decorator and component respectively

* 1. Story behind the demos
     1. Demo Name: CollectionAnd$SyncronizedCollection
        1. By our definition of Decorator SyncronizedCollection does NOT decorate Collection. The reason being that *equals* and *hashCode* are defined in the Collection interface but are not declared in SyncronizedCollection. However, SyncronizedCollection inherits these methods from Object. We felt this was a special case that did not fit under our requirement of decorating ALL methods of the component and thus still did not consider it a decorator.
     2. Demo Name: ClassNodeAndAncestors
        1. After looking at the ASM class hierarchy, we determined that the ancestors of ClassNode are ClassVisitor and Object. We do not consider ClassNode to decorate ClassVisitor because this falls under the category described above where the behavior of this would mimic that of a LinkedList. While ClassNode does implement all the methods of ClassVisitor, ClassVisitor has a field of itself. We determine that in this case ClassVisitor itself is not a decorator and neither are its subclasses.
     3. Demo Name: ProjectClassParserDecorator
        1. This demo shows our use of the Decorator pattern in our Class Diagram parsers.
     4. Demo Name: ProjectSequenceParserDecorator
        1. This demo shows our use of the Decorator pattern in our Sequence Diagram parsers.

Hard: Adapter

1. The essential syntactic qualities of Adapter are…
   * 1. If a class for every abstract method on its super…
        1. Has the exact same methods as its super
        2. Class’ implementation of method has the same owner as the current field
     2. Additionally, you can adapt an interfacae
        1. Since every method on the interface is abstract, if the Class’ implementation of that method (with the same name, args, return type) contains a call to a field on the Class.

… Then we say that that Class adapts that Field to a target interface or abstract class.

1. The difference between an adapter and a decorator is that the adapter bridges the gap between an incompatible class and interface. The decorator “decorates” an already existing function with its own special directions directly at runtime. So the purpose of an adapter is to call the methods of another class without changing them so that you can treat it like an object of an incompatible interface. The purpose of a decorator, however, is to *add* functionality at runtime. Additionally, unlike the decorator, the adadpter pattern would never adapt a class to its own interface. So if, for example, a class extended a class and had an instance of that class. That would be a decorator and not an adapter, because there’s no need to adapt that class.
2. Our group determines whether or not something is an adapter by…
   1. Analyzing each class in our diagram
   2. For each field in said class
   3. For every recursively gathered superclass and interface
   4. For every abstract method on the superclass or interface
   5. Check if…
      1. The class has a method with matching name, args, and return type of the current method of the superclass / interface
      2. The class’ implementation of the method has a call to the current field we’re checking for adaptation
   6. If both are true for every method, then this class is an adapter for that field with all the interfaces that these predicates evaluated to true
3. 1. JScheme adapter:
      1. The Jscheme example is an adapter, because it adapts the abstract *handle* method in the SchemeFunction interface. JSchemeFunction class has a JScheme object, implements the SchemeFunction interface’s handle method, and in that method it calls *eval* on its JScheme object. Thus, we say JSchemeFunction adapts JScheme to SchemeFunction
   2. InputStreamReader
      1. The InputStreamReader is *not* an adapter. InputStreamReader extends java.io.Reader, which has two abstract methods, read() and close(). InputStreamReader has a field of type StreamDecoder and calls methods on this object in both abstract methods. However, this is not the adapter method becauase StreamDecoder also extends java.io.Reader. Thus, there is no need to adapt StreamDecoder to Reader, because it already is one.
   3. MouseAdapter
      1. The MouseAdapter abstract class is *not* an adapter, simply because it does not have an instance of the classes it is supposed to adapt. Its adaptees are supposed to extend the Mouse Adapter class in order to be adapted to MouseListener. However, since they already extend MouseAdapter, which implements MouseListener, there’s no need to adapt them in the first place.
   4. Examples of adapter in our project:
      1. We have no examples of adapter in our project