Replace Partially Implemented Interface with Adapter

Your class implements an interface but only provides code for some of the interface's methods.

Move the implemented methods to an Adapter of the interface and make the Adapter accessible from a Factory Method.

```
public class CardComponent extends Container implements MouseMotionListener ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(this);
  }
  public void mouseDragged(MouseEvent e) {
    e.consume();
    dragPos.x = e.getX();
    dragPos.y = e.getY();
    setLocation(getLocation().x+e.getX()-currPos.x,
                getLocation().y+e.getY()-currPos.y);
    repaint();
  }
  public void mouseMoved(MouseEvent e) {
  }
```

```
public class CardComponent extends Container ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(createMouseMotionAdapter());
  }
  private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        e.consume();
        dragPos.x = e.getX();
        dragPos.y = e.getY();
        setLocation(getLocation().x+e.getX()-currPos.x,
                    getLocation().y+e.getY()-currPos.y);
        repaint();
      }
    };
  }
```

Motivation

Empty methods in concrete classes bother me. I often find that they're there because a class needs to satisfy a contract by implementing an interface, but only really needs code for *some* of the interface's methods. The rest of the methods get declared, but remain empty: they were added to satisfy a compiler rule. This may not bother you, but it bothers me. I find that these empty methods add to the heftiness of a class's interface (i.e. it's public methods), falsely advertise behavior (I'm a class that can, among other things, do X(), Y() and Z() – only I really only provide code for X()), and forces me to do work (like declaring empty methods) that I'd rather not do.

The Adapter pattern provides a nice way to refactor this kind of code. By implementing empty methods for every method defined by an interface, the Adapter lets me subclass it to supply just the code I need. In Java, I don't even have to formally declare an Adapter subclass: I can just create an anonymous inner Adapter class and supply a reference to it from a Factory Method.

Communication	Duplication	Simplicity
Empty methods on a class don't communicate very much at all. Either someone forgot to delete the empty method, or it is just there because an interface forces you to have it there. It is far better to communicate only what you actually implement, and an Adapter can make this feasible.	If more than one of your classes partially implements an interface, you'll have numerous empty methods in your classes. You can remove this duplication by letting each of the classes work with an Adapter which handles the empty method declarations.	It is always simpler to supply less code than more. This refactoring gives you a way to cut down on the number of methods your classes declare. In addition, when used to adapt multiple interfaces, it can provide a nice way to partition methods in each of their respective adapters.

Mechanics

1. If you don't already have an adapter for the interface (which we'll call A), use *Adapt Interface* to create one. Then create a Factory Method that will return a reference to an instance of your Adapter (which we'll call AdapterInstance).

2. Delete every empty method in your class that's solely there because your class implements A.

3. For those methods specified by A for which you have code, move each to your AdapterInstance.

4. Remove code declaring that your class implements A.

5. Supply the AdapterInstance to clients who need it.

Example

We'll use the example from the code sketch above. In this case we have a class called `CardComponent` that extends the JDK `Component` class and implements the JDK's `MouseMotionListener` interface. However, it only implements one of the two methods declared by the `MouseMotionListener` interface. Let's see how Adapter can improve the code.

1. The first step involved creating a Factory Method for our AdapterInstance. If we don't have an AdapterInstance, we need to create one using the refactoring, *Adapt Interface*. In this case, the JDK already supplies us with an adapter for the `MouseMotionListener` interface. It's called `MouseMotionAdapter`. So we create the following new method on the `CardComponent` class, using Java's handy anonymous inner class capability:

```
private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
    };
}
```

2. Next, we delete the empty method(s) that `CardComponent` declared because it implemented `MouseMotionListener`. In this case, it implemented `mouseDragged()`, but did not implement `mouseMoved()`.

```
    public void mouseMoved(MouseEvent e) {}
```

3. We're now ready to move the `mouseDragged()` method from `CardComponent` to our instance of the `MouseMotionAdapter`:

```
private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            e.consume();
            dragPos.x = e.getX();
            dragPos.y = e.getY();
            setLocation(getLocation().x+e.getX()-currPos.x,
                        getLocation().y+e.getY()-currPos.y);
            repaint();
        }
    };
}
```

4. Now we can remove the `implements MouseMotionListener` from `CardComponent`.

```
public class CardComponent extends Container implements MouseMotionListener {
```

5. Finally, we must supply the new adapter instance to clients that need it. In this case, we must look at the constructor. It has code that looks like this:

```
public CardComponent() {
   …
   addMouseMotionListener(this);
}
```

This needs to be changed to call our new, private, Factory Method:

```
public CardComponent() {
   …
   addMouseMotionListener(createMouseMotionAdapter());
}
```

Now we test. Unfortunately, since this is mouse related code, I don't have automated unit tests. So I resort to some simple manual testing and confirm that everything is ok.