

Workplan for design session #3-#5 (aka TD4-5) for students

This session be devoted to the extension of the V1 architecture to take the new user stories into account.

User stories, set #2

#6. As a user I can get an HTML description of my configuration, if it is complete and valid.

#7. As a user I can get the current price (in euros) of my configuration, even if it is incomplete (but it must be valid).

#8. As a user I can select the color of the exterior paint, from a limited subset provided by the application (the subset can depend on the exterior part type that I have selected).

Changes in the API

The new user stories show that a `PartType` does not contain enough information to deal with individual choices such as color. As explained before, and just like in e-commerce websites, a customer chooses a part type but get an instance of it, which is itself configurable through individual properties (e.g. color, size, flavor, etc.).

As a consequence, operations for getting elements of the current configuration must return an instance and not a type.

For the V2 version, we will use the new signatures defined below for `Configuration.java`

```
public Set<Part> getSelectedParts();
```

```
public Optional<Part> getSelectionForCategory(Category category);
```

and for `Part.java` :

```
public interface Part extends PropertyManager {  
    default String getName() {  
        return this.getClass().getTypeName();  
    };  
    Category getCategory();  
    PartType getType();  
}
```

Part class implementation

The principles behind the application of the type-instance patter in Java are the following ones:

- Specific attributes and behaviour are defined in subclasses of `PartImpl`, for instance the `EG100` class defines specific attributes or specific function such as pollution computation;
- The `PartTypeImpl` class is in charge of instantiating the proper class to create a `Part` implementation

Therefore in the implementation class of `PartType` (`PartTypeImpl`) we need a factory operation that instantiates the proper class, using a `Class<T>` attribute:

```
public class PartTypeImpl implements PartType {  
    private String name;  
    private Class<? extends PartImpl> classRef;  
    private Category category;  
    public PartTypeImpl(String name, Class<? extends PartImpl> classRef, Category category) {
```

```

        this.name = name;

        this.classRef = classRef;

        this.category = category;
    }

    public PartImpl newInstance() {
        Constructor<? extends PartImpl> constructor;

        try {
            constructor = classRef.getConstructor();

            return constructor.newInstance();

        } catch (Exception e) {
            Logger.getGlobal().log(Level.SEVERE, "constructor call failed", e);

            System.exit(-1);
        }

        return null;
    }
}

```

Print support (US #6)

Question #1 Extend the v1 architecture so that the API includes a new operation (printDescription) that takes a PrintStream stream as a parameter. Modify the implementation classes.

Question #2 Provide an object diagram (for the EG210, TSF7, XS, IS choices) and then a sequence diagram that shows the execution of this new operation.

Properties support (US #7 and #8)

The support of the user stories #7 and #8 will be provided by an extension of the API in the form of properties. For the v2 version of the configurator, a new interface will be added to the API types: PropertyManager. Here is the description in Java.

```

public interface PropertyManager {
    /**
     * Returns an immutable set of the property names supported by the property manager.
     *
     * @return
     */
    public Set<String> getPropertyNames();

    /**
     * Returns the immutable set of discrete string values for a given property.
     * For properties that have a non explicit set of possible values (eg double converted to strings),
     * or for a non existing property name, returns an empty set.
     *
     * @param propertyName a non-null string reference
     * @return an immutable set (see above)
     */
    public Set<String> getAvailablePropertyValues(String propertyName);

    /**

```

```

    * Returns the optional value of a property.
    * If the object does not support that property then an empty optional is returned.
    * @param propertyName the property to read
    * @return
    */
    public Optional<String> getProperty(String propertyName);

    /**
    * Sets the value of a given property.
    * If there is not such property, or if it not writable, or if the value is invalid
    * then an IllegalArgumentException is thrown.
    * @param propertyName
    * @param propertyValue
    * @throws IllegalArgumentException (see above)
    */
    void setProperty(String propertyName, String propertyValue);

}

```

All parts will support this interface.

Question #3 Study and compare the different ways to add an implementation for the PropertyManager in the v1 architecture.

Among the possibilities we choose to put the property management code in PartImpl. Below is a possible implementation of the property management (in PartImpl.java for instance).

```

public class PartImpl implements Part {

    private PartType type;

    private class Property {
        public final Supplier<String> getter;
        public final Consumer<String> setter;
        public final Set<String> possibleValues;

        Property(Supplier<String> getter, Consumer<String> setter, Set<String> possibleValues) {
            this.getter = getter;
            this.setter = setter;
            this.possibleValues = possibleValues;
        }
    }

    private Map<String, Property> properties = new HashMap<>();

    protected void addProperty(String name, Supplier<String> getter, Consumer<String> setter,
        Set<String> possibleValues) {
        properties.put(name, new Property(getter, setter, possibleValues));
    }

    @Override
    public Set<String> getPropertyNames() {
        return Collections.unmodifiableSet(properties.keySet());
    }

    @Override
    public Optional<String> getProperty(String propertyName) {
        Objects.requireNonNull(propertyName);
    }
}

```

```

        if (properties.containsKey(propertyName)) {
            return Optional.of(properties.get(propertyName).getter.get());
        }
        return Optional.empty();
    }

    @Override
    public void setProperty(String propertyName, String propertyValue) {
        Objects.requireNonNull(propertyName);
        Objects.requireNonNull(propertyValue);

        if ((properties.containsKey(propertyName)) && (properties.get(propertyName).setter != null)) {
            properties.get(propertyName).setter.accept(propertyValue);
        } else {
            throw new IllegalArgumentException("bad property name or value: " + propertyName);
        }
    }

    @Override
    public Set<String> getAvailablePropertyValues(String propertyName) {
        if (properties.containsKey(propertyName)) {
            return Collections.unmodifiableSet(properties.get(propertyName).possibleValues);
        }
        return Collections.emptySet();
    }
}

```

Question #4 Explain how the Command design pattern is implemented in the code above (hint: `Supplier<String>` is an interface that has only one operation, with no parameters and that returns a `String`; `Consumer<String>` is an interface with only one operation, which take one `String` parameter and returns nothing, and lambdas are nice).

Question #5 Define a Java `HelloWorld` class that has only one property, named `message`, which can only be read, and whose value is always "Hello world".

Question #6 Provide a class diagram and a sequence diagram for this class, that show the application of the Command design pattern (hint: it is applied twice here).

Question #7 Provide the Java code of `XS` class, subclass of `Exterior`, which has a property named `paintcolor` and that implements a getter and setter for this property (the set of possible values will be red and blue). Hint: an enumeration type for the color possibilities could be useful.