Step Builder pattern

Recently I wanted to implement a builder in my Grep4j API, but, as already happened in the past with the builder or other creational patterns, I was not completely satisfied with the result. A Builder pattern is a design you implement when you want to separate the creation of an object from its representation. For example, let say you have a Java Panino object composed of few ingredients:

```
package com.stepbuilder.bar;
import java.util.List;
public class Panino {

 private final String name;
 private String breadType;
 private String fish;
 private String cheese;
 private String meat;
 private List vegetables;

 public Panino(String name) {
  this.name = name;
 }

 public String getBreadType() {
  return breadType;
 }

 public void setBreadType(String breadType) {
  this.breadType = breadType;
 }

 public String getFish() {
  return fish;
 }

 public void setFish(String fish) {
  this.fish = fish;
 }

 public String getCheese() {
  return cheese;
 }

 public void setCheese(String cheese) {
  this.cheese = cheese;
 }

 public String getMeat() {
  return meat;
 }

 public void setMeat(String meat) {
  this.meat = meat;
 }

 public List getVegetables() {
  return vegetables;
 }

 public void setVegetables(List vegetables) {
  this.vegetables = vegetables;
 }

 public String getName() {
  return name;
 }

 @Override
 public String toString() {
  return "Panino [name=" + name + ", breadType=" + breadType + ", fish="
```

```
      + fish + ", cheese=" + cheese + ", meat=" + meat
      + ", vegetables=" + vegetables + "]";
  }


 }
```

Now, in order to create a Panino you can write your Builder class that, more or less, will look like the following.

```
package com.stepbuilder.bar;
import java.util.ArrayList;
import java.util.List;
public class PaninoBuilder {

 private String name;
 private String breadType;
 private String fish;
 private String cheese;
 private String meat;
 private List vegetables = new ArrayList();

 public PaninoBuilder paninoCalled(String name){
  this.name = name;
  return this;
 }

 public PaninoBuilder breadType(String breadType){
  this.breadType = breadType;
  return this;
 }

 public PaninoBuilder withFish(String fish){
  this.fish = fish;
  return this;
 }

 public PaninoBuilder withCheese(String cheese){
  this.cheese = cheese;
  return this;
 }

 public PaninoBuilder withMeat(String meat){
  this.meat = meat;
  return this;
 }

 public PaninoBuilder withVegetable(String vegetable){
  vegetables.add(vegetable);
  return this;
 }

 public Panino build(){
  Panino panino = new Panino(name);
  panino.setBreadType(breadType);
  panino.setCheese(cheese);
  panino.setFish(fish);
  panino.setMeat(meat);
  panino.setVegetables(vegetables);
  return panino;
 }
 }
```

A user will be then able to nicely build a Panino using this builder.

```
package com.stepbuilder.bar.client;
import com.stepbuilder.bar.Panino;
import com.stepbuilder.bar.PaninoBuilder;
public class Bar {

 public static void main(String[] args) {
  Panino marcoPanino = new PaninoBuilder().paninoCalled("marcoPanino")
    .breadType("baguette").withCheese("gorgonzola").withMeat("ham")
    .withVegetable("tomatos").build();

  System.out.println(marcoPanino);
 }
}
```

So far so good.
But what I don't like of the traditional Builder pattern is the following:

- It does not really guide the user through the creation.
- A user can always call the build method in any moment, even without the needed information.
- There is no way to guide the user from a creation path instead of another based on conditions.
- There is always the risk to leave your object in an inconsistent state.
- All methods are always available, leaving the responsibility of which to use and when to use to the user who did not write the api.

For instance, in the Panino example, a user could write something like this:

```
package com.stepbuilder.bar.client;
import com.stepbuilder.bar.Panino;
import com.stepbuilder.bar.PaninoBuilder;
public class Bar {

 public static void main(String[] args) {
  Panino marcoPanino = new PaninoBuilder().paninoCalled("marcoPanino")
    .withCheese("gorgonzola").build();

  // or
  marcoPanino = new PaninoBuilder().withCheese("gorgonzola").build();
  // or
  marcoPanino = new PaninoBuilder().withMeat("ham").build();
  // or
  marcoPanino = new PaninoBuilder().build();
 }
}
```

All the above panino instances are wrong, and the user will not know until runtime when he will use the Panino object.
You can put a validation in the build method of course, but still a user will be not able to recover from a badly builder usage.
You could also put default values around all the required properties, but then the readability of the code will be lost
(`newPaninoBuilder().build();` what are you building here?) and often you really need some input from the user (a user and password for a connection for example).

So here is my solution called Step builder, an extension of the Builder patter that fully guides the user through the creation of the object with no chances of confusion.

```
package com.stepbuilder.bar;
import java.util.ArrayList;
import java.util.List;
public class PaninoStepBuilder {
```

```
public static FirstNameStep newBuilder() {
        return new Steps();
}

private PaninoStepBuilder() {
}

/**
 * First Builder Step in charge of the Panino name.
 * Next Step available : BreadTypeStep
 */
public static interface FirstNameStep {
        BreadTypeStep paninoCalled(String name);
}

/**
 * This step is in charge of the BreadType.
 * Next Step available : MainFillingStep
 */
public static interface BreadTypeStep {
        MainFillingStep breadType(String breadType);
}

/**
 * This step is in charge of setting the main filling (meat or fish).
 * Meat choice : Next Step available : CheeseStep
 * Fish choice : Next Step available : VegetableStep
 */
public static interface MainFillingStep {
        CheeseStep meat(String meat);

        VegetableStep fish(String fish);
}

/**
 * This step is in charge of the cheese.
 * Next Step available : VegetableStep
 */
public static interface CheeseStep {
        VegetableStep noCheesePlease();

        VegetableStep withCheese(String cheese);
}

/**
 * This step is in charge of vegetables.
 * Next Step available : BuildStep
 */
public static interface VegetableStep {
        BuildStep noMoreVegetablesPlease();

        BuildStep noVegetablesPlease();

        VegetableStep addVegetable(String vegetable);
}

/**
 * This is the final step in charge of building the Panino Object.
 * Validation should be here.
 */
public static interface BuildStep {
        Panino build();
}

private static class Steps implements FirstNameStep, BreadTypeStep, MainFillingStep, CheeseStep, Vege

        private String name;
        private String breadType;
        private String meat;
        private String fish;
        private String cheese;
        private final List<String> vegetables = new ArrayList<String>();
```

```
                public BreadTypeStep paninoCalled(String name) {
                        this.name = name;
                        return this;
                }

                public MainFillingStep breadType(String breadType) {
                        this.breadType = breadType;
                        return this;
                }

                public CheeseStep meat(String meat) {
                        this.meat = meat;
                        return this;
                }

                public VegetableStep fish(String fish) {
                        this.fish = fish;
                        return this;
                }

                public BuildStep noMoreVegetablesPlease() {
                        return this;
                }

                public BuildStep noVegetablesPlease() {
                        return this;
                }

                public VegetableStep addVegetable(String vegetable) {
                        this.vegetables.add(vegetable);
                        return this;
                }

                public VegetableStep noCheesePlease() {
                        return this;
                }

                public VegetableStep withCheese(String cheese) {
                        this.cheese = cheese;
                        return this;
                }

                public Panino build() {
                        Panino panino = new Panino(name);
                        panino.setBreadType(breadType);
                        if (fish != null) {
                                panino.setFish(fish);
                        } else {
                                panino.setMeat(meat);
                        }
                        if (cheese != null) {
                                panino.setCheese(cheese);
                        }
                        if (!vegetables.isEmpty()) {
                                panino.setVegetables(vegetables);
                        }
                        return panino;
                }

        }
}
```

The concept is simple:

1. Write creational steps inner classes or interfaces where each method knows what can be displayed next.
2. Implement all your steps interfaces in an inner static class.
3. Last step is the BuildStep, in charge of creating the object you need to build.

The user experience will be much more improved by the fact that he will only see the next step methods available, NO build method until is the right time to build the object.

```
package com.stepbuilder.bar.client;
import com.stepbuilder.bar.Panino;
import com.stepbuilder.bar.PaninoBuilder;
import com.stepbuilder.bar.PaninoStepBuilder;
public class Bar {

 public static void main(String[] args) {
  Panino solePanino = PaninoStepBuilder.newBuilder()
        .paninoCalled("sole panino")
        .breadType("baguette")
        .fish("sole")
        .addVegetable("tomato")
        .addVegetable("lettece")
        .noMoreVegetablesPlease()
        .build();


 }
}
```

The user will not be able to call the method breadType(String breadType) before calling the paninoCalled(String name) method, and so on.
Plus, using this pattern, if the user choose a fish panino, I will not give him the possibility to add cheese (i'm the chef, I know how to prepare a panino).
In the end, I will have a consistent Object and user will find extremely easy to use the API because he will have very few and selected choices per time.

We could have different panino traditional builders, FishPaninoBuilder, MeatPaninoBuilder, etc, but still we would face the inconsistent problems and the user will still need to understand exactly what was your idea behind the builder.
With the Step Builder the user will know without any doubt what input is required, making his and your life easier.