**Design Patterns**

Miguel Real 55677

Note: For brevity, it’s to be assumed all packages/files mentioned henceforth are located in “src/main/java/org/jabref/”.

1. **Template Method Pattern (Behavioral)**

Package of origin: “logic/importer/fetcher/transformers/”

***Brief explanation of the found pattern:*** The abstract AbstractQueryTransformer class includes a transform() method with several implementations (each of which can be seen as independent instances of this pattern). Each of these utilize several abstract methods (*steps*) declared in the AbstractQueryTransformer class. It then falls upon the concrete classes in this package to implement each step of the algorithm in their own way in order to produce a functional algorithm when the transform() method is called.

This is a textbook example of the Template Method pattern, which is based on a method (the *template*) in an abstract class that defines an algorithm by calling other methods (the *steps*). These methods (usually abstract, though not always) will then be implemented by subclasses in order to tailor the algorithm to their needs.

*One of the implementations of the transform() method. Several of the methods called (handleAuthor(), handleTitle(), handleJournal(), etc.) are defined as abstract in this class and are then implemented in subclasses.*

*Here we can see a concrete implementation of a step, handleAuthor(), from ScholarQueryTransformer.java:*

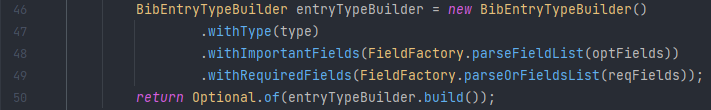
1. **Builder Pattern (Creational)**

Package of origin: “model/entry/”

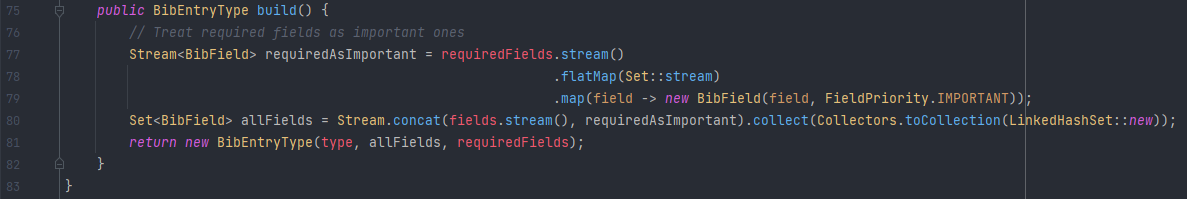
***Brief explanation of the found pattern:*** We can observe a Builder pattern in the interactions between the BibEntryType (*Product*), BibEntryTypeBuilder (*Builder*), and BibEntryTypesManager (*Director*) classes. In the parse() method of the latter class a builder object is created and given constraints on the product it will build. A product is then returned by calling the builder class’s build() method, which returns a BibEntryType with the characteristics set down by the director class previously.

Although this behaviour is consistent with the Builder pattern, it is not a textbook implementation of it, as that would involve a slightly different approach, with a global Builder variable in the Director, a build/make method in the Director which given the desired product characteristics would call the necessary component-building methods of the builder class (using the aforementioned builder variable), and a builder class whose only purpose in the pattern is to hold the operations responsible for the creation of a product’s component parts.





*parse() method of the BibEntryTypesManager class, abridged to only include code relevant to the pattern in discussion (lines 46-50). A builder class is created and given directions on the product it will build.*

*Excerpt from the BibEntryTypeBuilder class, the methods seen are a sample of the various existing methods to constrain the characteristics of the product that will be created.*



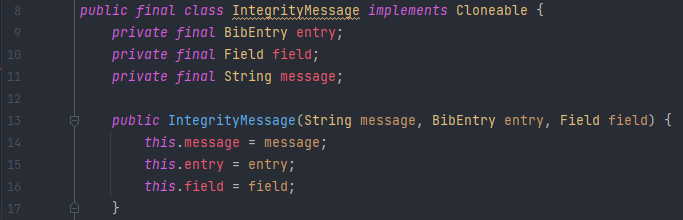
*build() method from BibEntryTypeBuilder. This method (usually part of the Director class) will create a Product based on the characteristics dictated by the Director.*

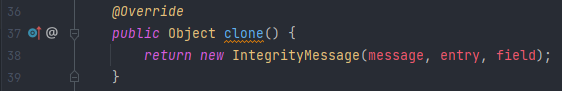
1. **Prototype Pattern (Creational)**

File of origin: “logic/integrity/IntegrityMessage.java”

***Brief explanation of the found pattern:*** IntegrityMessage is a small class focused around 3 private variables that store its characteristics (entry, field, and message). The constructor is used as a setter, receiving 3 parameters that correspond to the 3 variables. It includes some relatively unremarkable methods, like getter methods for the 3 variables and toString(), equals() and hashCode() methods which do what one would expect. More interestingly though, it also includes a clone() method, which is the one relevant for this design pattern, its purpose is to create another instance of the IntegrityMessage class with the exact same fields as the current one (a clone).

This is a textbook implementation of the Prototype pattern, which is defined by an object having a clone() method whose function is to create another instance of the same class with identical field values, essentially a clone (as suggested by the method’s namesake).

*The 3 main variables of the IntegrityMessage class along with its constructor (which works as a setter for the aforementioned variables).*



*clone() method of the IntegrityMessage class, its purpose is to create another instance of IntegrityMessage with the exact same fields as the current one (a clone).*