1 Problem analysis

1.1 Concepts

Table 1 shows an overview of all the concepts we identified in the problem domain.

1.2 Actions

Table 2 shows an overview of all the actions we identified in the problem domain.

1.3 Rules

Table 3 shows an overview of all the rules we found in the problem domain that are applicable to the concepts discussed in Table 1. Table 4 shows an overview of all the rules we found in the problem domain that are applicable to the actions discussed in Table 2.

1.4 Responsibilities

Table 5 shows an overview of all the responsibilities of the concepts we discussed in Table 1. Table 6 shows an overview of all the responsibilities of the actions we discussed in Table 2.

1.5 Constraints

The application has the following constraint:

- The application does not support a user-interface.

2 Design

In this section, we discuss the whole class diagram by dividing it in four parts.

2.1 The MVC part

Figure **??** shows the class diagram of the Model-View-Controller (MVC) part. We chose this pattern despite the fact it is mostly used when dealing with user-interfaces and the application having the constraint it currently does not support a user-interface. The reason why we chose it anyway is four-fold:

- One of the responsibilities of the application is to display the references in the right style. A simple way to do this is to use a small preview window to display the references in. This is something that would be done by the View.

- A user-interface may be necessary to support future feature requests.

- MVC provides a clear separation between the presentation logic (View), business logic (Model) and control flow logic (Controller).

Table 1: An overview of all the concepts in the problem domain.

Concept	Description
Kaiburr crystal	A Kaiburr crystal powers a light saber. It has the following properties: • Color: The crystal's color. • Name: The crystal's name. • Power usage: The crystal's power usage. • Harvested from: The planet where the crystal was harvested. • Price: The crystal's price.
Force user	A force user can use the Force. A force user has the following properties: • Force: The amount of force the force user has. • Age: The force user's age. • Title: The force user's title.
Crystal harvest job	A crystal harvest job has the following properties: • Planet: The planet where a crystal can be harvested. • Harvest price: The price for harvesting a crystal.
Product	A product has the following properties: • ID: An unique identification number that identifies the product. • Name: The product's name.
Light saber	A light saber is a Product and thus shares the same properties as the Product concept. A light saber has the following properties: • ID: An unique identification number that identifies the light saber. • Name: The light saber's name. • Kaiburr crystal: The Kaiburr crystal that powers the light saber.
Store inventory	The store inventory has the following properties: • Product: The product that is in stock. • Stock: The quantity of this product that are in stock.
Order	An order has the following properties: • ID: An unique identification number that identifies the order. • Date: The date and time of when the order was placed. • Force user: The force user that placed the order.
Order line	An order line has the following properties: • Order: The order this order line belongs with. • Product: The product that is ordered. • Quantity: The quantity of products that is ordered. • Unit price: The price of one unit of the product at the time the product was ordered.

Table 2: An overview of all the actions in the problem domain.

Action	**Description**
Application start-up	When the application starts, it reads three parameters that were provided by the user: (1) a path to an XML file containing a list of sabers, (2) the age of the force user, (3) the name or id of a saber. The application prints the following information: (1) the force user's force, (2) the force that is used by the light saber's crystal, (3) the name of the light saber's crystal, (4) the price of the light saber's crystal.
Load new light sabers	A Jedi master can load new light sabers in the web shop.
Query light sabers	A Jedi master can query the light sabers and stock.
Order light saber	A Padawan can order a saber.
Archive orders	The application archives all the orders for the Jedi master.

Table 3: An overview of all the rules in the problem domain that are applicable to the concepts discussed in Table 1.

Concept name	**Applicable rules**
Kaiburr crystal	• The crystal's price is equal to $Cr * F$, where Cr is the price for harvesting the crystal, and F is the crystal's power usage.
Force user	• The force user's title is selected depending on the force user's force value. The title Padawan is selected when the force value is less than 93,2. The title Jedi master is selected when the force value is equal to or greater than 93,2. • The force user's force is equal to $A * 10$, where A is the force user's age. After the age of 18, the force is unlimited.

Table 4: An overview of all the rules in the problem domain that are applicable to the actions discussed in Table 2.

Action name	**Applicable rules**
Check reference for completeness	• The *Reference type* determines which *Fields* are required for a complete reference.

Table 5: An overview of all the responsibilities of the concepts discussed in Table 1.

Name	**Responsibilities**
Reference	• A *Reference* can provide its data. • A *Reference* can provide the essential *Fields* in the correct order for displaying the reference.
Fields	• A *Field* can provide its data.
Reference type	• A *Reference type* can correctly order the *Fields.* • A *Reference type* can determine if all essential *Fields* in a *Reference* are present and have a value.
Reference style	• A *Reference style* can apply the correct style on the value of a *Field.*

Table 6: An overview of all the responsibilities of the actions discussed in Table 2.

Name	**Responsibilities**
Read references	• Uses the correct *Storage format* to read the *References.* • Provides all the read *References* of which the *Reference type* is known.
Save references	• Retrieves the data from the *References* and uses the *Storage format* to save the *References.*
Display references	• Checks if the *References* are complete. • Retrieves the *Fields* and *Reference type* from the *References* and displays this data in the correct order according to the *Reference type* and correct style according to the *Reference style.*
Check references for completeness	• Retrieves the *Fields* and *Reference type* from every *Reference* and checks if the *Fields* are complete according to the *Reference type.* • Provides whether or not the *References* are complete.
Add free field to reference	• Checks if the name of the *Free field* is different from the names of the other *Fields* in the *Reference.* If so, the *Free field* is added to the *Reference.*
Set reference style	• Sets a new *Reference style.*

- The Model, View and Controller objects have distinct responsibilities and can be changed independent of each other. So when a user-interface is needed, only the existing View needs to be changed (unless more functionality is added with this new View meaning at least the Controller needs to be changed as well).

In the class diagram in Figure **??**, the Model, Controller and View interfaces are solely used to indicate which classes behave like Models, Controllers and Views. The MVC-related abstract classes contain methods to support most of the actions we discussed in Table 2. They are responsible for:

- Reading references from a source.
- Saving references in a specific *Storage format.*
- Displaying references in an appropriate *Reference style.*
- Setting a (new) *Reference style.*
- Adding *Free fields* to references.

The methods in the MainController class request the MainModel class to execute the required actions. The MainModel uses other classes to execute the required actions (see the upcoming sections). The only action the MainModel does self is setting the *Reference style.* In the case of displaying the references, the MainModel class returns the resulting references string representation to the MainController class and the MainController class passes this to the MainView class which will display it in a window. The MainView class is thus responsible for displaying the *References.*

The MVCFactory abstract class is responsible for creating the related Model, Controller and View objects. When the Controller object is created, it uses the MVCFactory class to create the related Model and View objects.

The ReferenceStyle enumeration is used by the MainModel and MainController classes to set a new *Reference style* or to display the references in the style indicated by the value of the ReferenceStyle. The reasons why we use an enumeration is because it restricts the possible values to indicate the different *Reference styles* to the ones defined in the enumeration and it can be used in switch statements.

2.2 The *References* and *Reference type* part

Figure **??** shows the class diagram of the *Reference* and *Reference type* part. The abstract classes and interfaces have the following responsibilities:

- The IField interface is responsible for providing the data of the *Field.*
- The IReference interface is responsible for providing the data of the *Reference*, adding a *Free field* to the *Reference*, checking if the the *Reference* has all the required *Fields* with a value according to the RequiredFieldsStrategy interface, and ordering the *Fields* according to the OrderFieldsStrategy interface.

- The RequiredFieldsStrategy interface is responsible for checking if a *Reference* is complete by looking if it contains all the essential *Fields* according to the *Reference type*.
- The OrderFieldsStrategy interface is responsible for ordering the *Fields* in a particular way according to the *Reference type*.
- The ReferenceTypeStrategyFactory abstract class is responsible for creating the appropriate RequiredFieldsStrategy and OrderFieldsStrategy classes for the given *Reference type*.

The Field and Reference objects are created when the references are read from a source (see Section 2.4). The ReferenceTypeStrategyFactory abstract class is used by the MainModel class to create the appropriate RequiredFieldsStrategy object and OrderFieldsStrategy object for the Reference object depending on its *Reference type*. These objects are used when the MainModel class wants to display the references.

One difference in this class diagram and the responsibilities discussed in Table 5 is that the Reference class as opposed to a ReferenceType class is responsible for (a) checking if the the *Reference* has all the required *Fields* with a value and (b) ordering the *Fields*. We chose this design because it makes more sense to let the Reference class work directly on its own Field objects as opposed to a different ReferenceType class that needs to get access to these Field objects. This would make the design needlessly more complex.

The ReferenceType enumeration is contained in the Reference class to indicate the *Reference type* of the *Reference*. It is also used by the ReferenceTypeStrategyFactory abstract class to create the appropriate RequiredFieldsStrategy and OrderFieldsStrategy classes according to the value of the ReferenceType. The reasons why we use an enumeration is because it restricts the possible values to indicate the different *Reference types* to the ones defined in the enumeration and it can be used in switch statements.

2.3 The *Reference style* part

Figure **??** shows the class diagram of the *Reference style* part. The abstract classes and interfaces have the following responsibilities:

- The IField interface is responsible for providing the data of the *Field* (see Section 2.2).
- The ReferenceStyleStrategy interface is responsible for applying the appropriate style on the value of the *Field* class depending on the name of the *Field* class.
- The ReferenceStyleFactory abstract class is responsible for creating the appropriate ReferenceStyleStrategy strategy for the given *Reference style*.
- The StyleStrategy interface is responsible for applying the appropriate style on a String value depending on the chosen Style value.
- The StyleFactory abstract class is responsible for creating the appropriate StyleStrategy class for the given Style value.

The ReferenceStyleFactory abstract class is used by the MainModel class to create the appropriate ReferenceStyleStrategy object. This object is used when the MainModel class wants to display the references. The StyleFactory abstract class is used by the ReferenceStyleStrategy concrete classes to create the appropriate StyleStrategy object. This object is used when these classes want to apply a specific style to the value of a *Field* object.

The ReferenceStyle enumeration is used by the MainModel class when it wants to set a new *Reference style* or display the references in the style indicated by the value of the ReferenceStyle. This enumeration is also used by the ReferenceStyleFactory abstract class to create the appropriate ReferenceStyleStrategy class according to the value of the ReferenceStyle. The reasons why we use an enumeration is because it restricts the possible values to indicate the different *Reference styles* to the ones defined in the enumeration and it can be used in switch statements.

2.4 The read and write part

Figure **??** shows the class diagram of the read and write part. The abstract classes and interfaces have the following responsibilities:

- IReference, see Section 2.2.
- The Reader interface is responsible for reading the references from a source of a certain *Storage format*.
- The Writer interface is responsible for saving the references in a specific *Storage format*.
- The AccessorFactory abstract class is responsible for creating the appropriate Reader and Writer classes depending on the *Storage format* which can be extracted from the String parameter.

The AccessorFactory abstract class is used by the MainModel class to create the appropriate Reader object and Writer object. These objects are used when the MainModel class wants to read or save the references.

3 Design patterns

When we created the design, we looked the patterns that were known to us at that time. We list these patterns and the problem they solve in Table 7.

We compared the problems we came across while we were designing, with the problems that each of the patterns in Table 7 solve. This way we could choose the most effective pattern which would solve our problem.

3.1 Reading and writing references

The first problem we came across was that we needed to create objects that could read and write references from and to a specific *Storage format*. The second problem was that the implementation of reading and writing from and to this format varies per format.

Table 7: The patterns that were known to us at the time we created the application design.

Name	**Category**	**Problem**
Facade	Structural	You need to use only a subset of a complex system or interact with the system in a particular way.
Adapter	Structural	A system has the right data and behavior but the wrong interface.
Bridge	Structural	The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.
Decorator	Structural	The object you want to use need to add some additional functionality, occurring before or after its base functionality.
Strategy	Behavioral	The selection of an algorithm that needs to be applied depends on the client making the request or the data being acted on. If you have a rule in place that does not change, this pattern is not needed.
Abstract Factory	Creational	Families of related objects need to be instantiated.

We identified the Strategy pattern as the only candidate to solve the second problem. The problem this pattern solves perfectly resembles our problem. We use this pattern as follows:

- The MainModel class corresponds to the Context class of the generic structure of the Strategy pattern.
- The Reader and Writer interfaces correspond to the Strategy abstract class of the generic structure of the Strategy pattern.
- The BibTexReader, DatabaseReader, EndNoteReader, XMLReader and the BibTexWriter, DatabaseWriter, EndNoteWriter and XMLWriter classes correspond to the ConcreteStrategy classes of the generic structure of the Strategy pattern.

We identified the Abstract Factory as the only candidate to solve the first problem. This pattern can be used to create the related Reader and Writer objects depending on the *Storage format*. We use this pattern as follows:

- The MainModel class corresponds to the Client class of the generic structure of the Abstract Factory pattern.
- The AccessorFactory abstract class corresponds to the AbstractFactory abstract class of the generic structure of the Abstract Factory pattern.
- The BibTexAccessorFactory, DatabaseAccessorFactory, EndNoteAccessorFactory and XMLAccessorFactory classes correspond to the ConcreteFactory classes of the generic structure of the Abstract Factory pattern.

- The Reader and Writer interfaces correspond to the AbstractProduct classes of the generic structure of the Abstract Factory pattern.
- The BibTexReader, DatabaseReader, EndNoteReader, XMLReader and the BibTexWriter, DatabaseWriter, EndNoteWriter and XMLWriter classes correspond to the Product classes of the generic structure of the Abstract Factory pattern.

Like we said earlier, there are not any alternative patterns that we could identify to solve these problems.

3.2 Ordering *Fields* and checking *References* for completeness

The third problem we came across was that we needed a way to order the *Fields* of a *Reference* depending on the *Reference type* when we want to display the reference. The value of this *Reference type* can likely be changed in run-time in the future.

We identified the Strategy pattern as the only candidate to solve this problem. The problem this pattern solves perfectly resembles our problem. We use this pattern as follows:

- The Reference class corresponds to the Context class of the generic structure of the Strategy pattern.
- The OrderFieldsStrategy interface corresponds to the Strategy abstract class of the generic structure of the Strategy pattern.
- The BookOrderFieldsStrategy, JournalArticleOrderFieldsStrategy, ProceedingsOrderFieldsStrategy and ThesisOrderFieldsStrategy classes correspond to the ConcreteStrategy classes of the generic structure of the Strategy pattern.

The fourth problem we came across was that we needed a way to check if a *Reference* is complete by checking if the reference has the required *Fields* depending on the *Reference type*. The value of this *Reference type* can likely be changed in run-time in the future.

We identified the Strategy pattern as the only candidate to solve this problem. The problem this pattern solves perfectly resembles our problem. We use this pattern as follows:

- The Reference class corresponds to the Context class of the generic structure of the Strategy pattern.
- The RequiredFieldsStrategy interface corresponds to the Strategy abstract class of the generic structure of the Strategy pattern.
- The BookRequiredFieldsStrategy, JournalArticleRequiredFieldsStrategy, ProceedingsRequiredFieldsStrategy and ThesisRequiredFieldsStrategy classes correspond to the ConcreteStrategy classes of the generic structure of the Strategy pattern.

Now that we found these solutions, we needed a way to create the appropriate strategies for the *Reference* depending on the *Reference type*. We identified the Abstract Factory pattern as the only pattern that could solve this fifth problem. We use this pattern as follows:

- The MainModel class corresponds to the Client class of the generic structure of the Abstract Factory pattern.
- The ReferenceTypeStrategyFactory abstract class corresponds to the AbstractFactory abstract class of the generic structure of the Abstract Factory pattern.
- The BookReferenceTypeStrategyFactory, JournalArticleReferenceTypeStrategyFactory, ProceedingsReferenceTypeStrategyFactory and ThesisReferenceTypeStrategyFactory classes correspond to the ConcreteFactory classes of the generic structure of the Abstract Factory pattern.
- The RequiredFieldsStrategy and OrderFieldsStrategy interfaces correspond to the AbstractProduct classes of the generic structure of the Abstract Factory pattern.
- The BookRequiredFieldsStrategy, JournalArticleRequiredFieldsStrategy, ProceedingsRequiredFieldsStrategy, ThesisRequiredFieldsStrategy and the BookOrderFieldsStrategy, JournalArticleOrderFieldsStrategy, ProceedingsOrderFieldsStrategy, ThesisOrderFieldsStrategy classes correspond to the Product classes of the generic structure of the Abstract Factory pattern.

Like we said earlier, there are not any alternative patterns that we could identify to solve these problems.

3.3 Display *References* in a *Reference style*

The sixth problem we came across was that we needed a way to display the values of the *Fields* of the *References* in a particular *Reference style*. This style can be changed in run-time.

We identified the Strategy pattern as the only candidate to solve this problem. We use this pattern as follows:

- The MainModel class corresponds to the Context class of the generic structure of the Strategy pattern.
- The ReferenceStyleStrategy interface corresponds to the Strategy abstract class of the generic structure of the Strategy pattern.
- The APAReferenceStyleStrategy, TurabianReferenceStyleStrategy and ChicagoReferenceStyleStrategy classes correspond to the ConcreteStrategy classes of the generic structure of the Strategy pattern.

The ConcreteStrategy classes check the names of the *Fields* in order to select the right style to apply to the values of the *Fields*. Each style changes the appearance of the values in a different way. So as the seventh problem, we needed a way to change the appearance of the values of the *Fields* according to the chosen style.

We identified the Strategy pattern as the only candidate to solve this problem. We use this pattern as follows:

- The APAReferenceStyleStrategy, TurabianReferenceStyleStrategy and ChicagoReferenceStyleStrategy classes correspond to the Context class of the generic structure of the Strategy pattern.
- The StyleStrategy interface corresponds to the Strategy abstract class of the generic structure of the Strategy pattern.
- The RegularStyleStrategy, BoldStyleStrategy and ItalicsStyleStrategy classes correspond to the ConcreteStrategy classes of the generic structure of the Strategy pattern.

Now we needed a way to create the appropriate strategy depending on the style that needed to be applied. We identified the Abstract Factory pattern as the only pattern that could solve this eight problem. We use this pattern as follows:

- The APAReferenceStyleStrategy, TurabianReferenceStyleStrategy and ChicagoReferenceStyleStrategy classes correspond to the Client class of the generic structure of the Abstract Factory pattern.
- The StyleFactory abstract class corresponds to the AbstractFactory abstract class of the generic structure of the Abstract Factory pattern.
- The ItalicsStyleFactory, BoldStyleFactory and RegularStyleFactory classes correspond to the ConcreteFactory classes of the generic structure of the Abstract Factory pattern.
- The StyleStrategy interface corresponds to the AbstractProduct class of the generic structure of the Abstract Factory pattern.
- The RegularStyleStrategy, BoldStyleStrategy and ItalicsStyleStrategy classes correspond to the Product classes of the generic structure of the Abstract Factory pattern.

And finally, we needed a way to create the appropriate strategy to apply the correct style depending on the *Reference style*. We identified the Abstract Factory pattern as the only pattern that could solve this problem. We use this pattern as follows:

- The MainModel class corresponds to the Client class of the generic structure of the Abstract Factory pattern.
- The ReferenceStyleFactory abstract class corresponds to the AbstractFactory abstract class of the generic structure of the Abstract Factory pattern.
- The APAReferenceStyleFactory, TurabianReferenceStyleFactory and ChicagoReferenceStyleFactory classes correspond to the ConcreteFactory classes of the generic structure of the Abstract Factory pattern.
- The ReferenceStyleStrategy interface corresponds to the AbstractProduct class of the generic structure of the Abstract Factory pattern.

- The APAReferenceStyleStrategy, TurabianReferenceStyleStrategy and ChicagoReferenceStyleStrategy classes correspond to the Product classes of the generic structure of the Abstract Factory pattern.

Like we said earlier, there are not any alternative patterns that we could identify to solve these problems.

3.4 Client

One of the final problems we came across was how we would let the client use the functionality of the system.

We could not identify any patterns from Table 7 to solve this problem. So we looked further and came across the Model-View-Controller pattern. The Controller could be used as an interface for the client to communicate with the system. The Controller forwards the request to the Model which will execute the required actions and return the results to the Controller. The Controller then notifies the View to visually display the results. We use this pattern as follows in our current design:

- The MainModel class corresponds to the Model class of the generic structure of the MVC pattern.
- The MainController class corresponds to the Controller class of the generic structure of the MVC pattern.
- The MainView class corresponds to the View class of the generic structure of the MVC pattern.

An alternative solution could be to have one class to act as the interface for the client to communicate with the system and also process and show the results of the requests. This would achieve the same thing with fewer objects. However, the MVC pattern has more advantages over this alternative solution in terms of flexibility and complexity:

- A user-interface may be necessary to support future feature requests. The MVC pattern deals specifically with structuring the applications with user-interfaces whereas the alternative solution does not.
- MVC provides a clear separation between the presentation logic (View), business logic (Model) and control flow logic (Controller). The one class in the alternative solution would be responsible for all this logic which would make it much more complex.
- The Model, View and Controller objects have distinct responsibilities and can be changed independent of each other. So when a user-interface is needed, only the existing View needs to be changed (unless more functionality is added with this new View meaning at least the Controller needs to be changed as well). The alternative solution does not assure that the responsibilities concerning the presentation, business and control flow logic can be changed independent of each other because they are all contained in the same object. This again increases the complexity and decreases the flexibility of this solution.

The final problem we came across is how to create the appropriate Model, View and Controller classes. We identified the Abstract Factory pattern as the only creational pattern that could solve this problem. We use this pattern as follows:

- The MainModel class corresponds to the Client class of the generic structure of the Abstract Factory pattern (an external object that wishes to use this system also corresponds as the Client).
- The MVCFactory abstract class corresponds to the AbstractFactory abstract class of the generic structure of the Abstract Factory pattern.
- The MainMVCFactory class corresponds to the ConcreteFactory class of the generic structure of the Abstract Factory pattern.
- The Model, Controller and View interfaces correspond to the Abstract-Product classes of the generic structure of the Abstract Factory pattern.
- The MainModel, MainController and MainView classes correspond to the Product classes of the generic structure of the Abstract Factory pattern.

4 Design decisions

In Section 2 we already briefly discussed the design decisions we made. In this section we will argue why they are the best decisions with respect to flexibility and complexity.

4.1 Enumerations

There are several occurrences of enumeration objects in the design:

- The *Reference type* enumeration to indicate the specific *Reference type* (Book, Journal article, etc.).
- The *Reference style* enumeration to indicate a specific *Reference style* (APA, Turabian, etc.).
- The Style enumeration to indicate a specific style (italics, bold, etc.).

The reasons why we use the enumeration objects over other objects (such as String) is because they restrict the possible values to the ones defined in the enumeration and they can be used in switch statements. The enumeration objects are used by the Abstract Factory classes to create an appropriate class depending on the value of the enumeration. This is where the switch statements are used with the enumeration objects to decide which classes will be created when. It provides an overview with minimal complexity. If more values need to be supported, they simply have to be added to the specific enumeration. So enumerations are also flexible.

4.1.1 Regarding the *Storage format*

In an earlier design we also used an enumeration object to indicate the *Storage format* (EndNote, XML, etc.). This object would be used by an Abstract Factory object to create the appropriate Reader and Writer objects depending on the *Storage format.* Later on we removed the enumeration object because the format can already be extracted from the String path that points to the specific source. For instance, if a user wants to read the references from a BibTex file, the *.bib* file extension already indicates the format. So a separate enumeration object does not need to be created, making the overall system less complex.

4.2 *Reference type*

Another important design decision we made was about how we would implement the *Reference type* for a *Reference.*

The responsibilities we discussed in Table 5 indicated that the *Reference type* itself should be responsible for (a) checking if the corresponding *Reference* has all the required *Fields* with a value and (b) ordering the *Fields.* This means a separate ReferenceType object should be defined that does these things. We ended up choosing a different design and let the Reference object be responsible for this. It makes more sense to let the Reference class work directly on its own Field objects as opposed to a different ReferenceType class that needs to get access to these Field objects. This would make the design needlessly more complex.

Another alternative solution to solve this problem would be to implement derivations of the Reference object which override the "check" and "order" abstract methods. These derivations contain the correct implementation of these methods for the relevant *Reference type* of the Reference. However, it is likely that the *Reference type* of a Reference can be changed in the future. So the Reference object would then have to be re-instantiated as a new Reference object using the new corresponding *Reference style* derivation. This is as inflexible as it can get.

For these reasons we went with our current design where the Reference object is responsible for these two things and delegates the exact implementation to two different strategies. This maximizes the flexibility.

5 Future changes

Our design is flexible enough to easily support many different changes. In this section we discuss an example of one of such changes. We also discuss an example that is much more complicated to support.

5.1 Simple future change

An example of a simple future change would be when the system needs to support a new *Reference style*: for instance MLA. We would have to take the following steps to implement this:

- Add the MLA value to the ReferenceStyle enumeration.

- Create a new concrete ReferenceStyleStrategy class for the MLA reference style: MLAReferenceStyleStrategy.
- Create a new concrete ReferenceStyleFactory class for the MLA reference style: MLAReferenceStyleFactory. This class will instantiate the newly created concrete MLAReferenceStyleStrategy class.
- Modify the getFactory method of the ReferenceStyleFactory abstract class so that the newly created MLAReferenceStyleFactory class can be created.

5.2 Complex future change

An example of a complex future change would be when the system needs to read more than just references from various *Storage formats*: for instance the document name where the references are used in. We assume the document name is also included in the same file or database as the references. We would have to take the following steps to implement this:

- Create a Document class which holds the name of the document and the list of references.
- Modify the MainModel class so it holds a reference to the Document class instead of the references. Also modify all the methods that worked directly with the references so they retrieve the references from the Document class first.
- Modify the Reader and Writer interfaces so that they use the Document object instead of the list of references.
- Modify all the concrete classes of the Reader and Writer interfaces in such a way so that they also read and write the document name.

As you can see, many more classes need to be modified compared to the other future change example. This future change requires a change in the design of the system because a whole new concept (Document) is introduced. For these reasons this future change is much more complex than the other one.