Report assignment 1

IM0102171811
Ivo Willemsen
851926289

Part 1: Problem analysis

Introduction

An application needs to be designed that enables a user to manage **references** and generate reference. References are stored in a **data store** (for example database or file**).** So the application needs to be able to read and save to the data store. The information in the data store is arranged according to a certain **storage format** (for example bibtex, EndNote or XML).

So at a use-case level, the following functionalities can be identified:

Use-case	Actor
Read references from data store	Client
Save references to data store	Client
Generate references	Client

References are managed within the context of a document. Imagine a user working on a document in TexMaker. The contents of the document will be stored in one file, and the references that are used by the document are stored in a different file. The use-cases that are mentioned above will adhere to the context of a document. When saving references to the data store, all the references that "belong" to the document will be stored in one unit of work, by using an identifier that is linked with the document (assumption).

Entities and attributes

The following entities can be identified by reading the case description. This section doesn't focus on responsibilities and actions, it only focusses on the identification of entities and attributes.

Client

That's the application that initiates the use-cases on behalf of the user. The client has on attribute:

1. The **storage format**. The storage format that will be used upon saving the references (see later). Per session, the client can choose the storage format that that will be used to convert the references

to the desired storage format when the references are saved to the data store

Reference

A reference consists of the following common concepts:

1. A label which serves as an interface. A document can identify the reference to be included in the document by using the label.
2. A list of fields. Each field consists of a name and value. According to the case description ("…. and so forth."), it can be deduced, that there are no predefined set of mandatory fields. It would not flexible to choose reference fields like "author", "title" and "booktitle". For that reason, this construction is kept flexible and no predefined set of reference fields is chosen
3. A type. A reference is of a certain type (for example book, journal article, paper in proceedings, thesis, etcetera)

The case description mentions that the difference tween one type of reference and another, is the name of the type and which fields are considered when the tool checks whether the references is complete.

I think it's only the name of the type of the reference, the fields don't matter. Take the example of two types of references with different names that have the same fields. The type of the references determines the order in which the fields are displayed. That can be different in both situations, but that doesn't depend on the fields, it depends on the type of the reference. So in this situation, what makes one type different than the other is only the name of the type. You could have two different types of references with a different name, but same fields and same rule for ordering, but they are still two different types.

Field

A field consists of the following common characteristics:

1. A field has a name and a value
2. The case description mentions several constraints regarding fields. Different terms are used that are a bit vague ("generated", "printed", "should have a value", "free", "completeness "). I will take some time to elaborate on these aspects (to create uniformity), as they are probably important for the design that follows. Of the list of fields that exist in the list, with respect to the generation (printing , mandatory) of the reference list, there are either mandatory fields (I will use the adjective "**mandatory**" going forward) or non-mandatory (I will use the adjective "**free**" going forward) fields. It can be extracted from the case description that a field is either a mandatory (which will be generated) field or a free field; there are no other types of fields
3. A field has a display format, but this is not an attribute of the field itself, it's a derived attribute that depends on the style. Also

a display format comes into play only during the generation phase. It's of no importance during the reading and saving of references (and fields) to and from the data store

4. A field occupies a certain slot in the order in the list of generated references. Again, this is not a property of the field itself, but depends on the type of the reference

Data store

The case mentions that the references will be stored in a either a database or a file. The concept is called **data store**, and examples of a data store are files and databases, or perhaps even a web service. This seems to suggest that there must be some option (client initiated action?, general configuration in file?) that decides which type of data store to use, although the case description doesn't mention it. But it's good to identify the existence of more than just one data store type, as this could impact the design at later times when this design needs to be made more flexible! Therefore, the assumption is made that the application should cater for the possibility of the user switching between multiple types of data stores in a flexible way.

Storage format

References are stored according to a certain **storage format**. Examples of formats are bibtex, EndNote and XML. Likewise as with the data store, the storage format will be a selectable feature. The user of the application will be able to switch from one storage format to the other (assumption).

Style

A style consists of the following common characteristics:

1. A style name
2. A **list of fields** with **display formats.** A style determines how certain fields are displayed. So this means that each style must know the fields that are displayed. Knowing how fields must be displayed means that there is a notion of fields that belong to the style. And for every field, an enumeration must be available (italic, bold, normal). So a style has a list of fields, and for every field a **display format** must be specified (italic, bold, normal)

A configured style affects all the references that are being managed at a certain moment of time. It is assumed that the user can switch the configured style at any moment in time.

Actions

From the use-cases introduction, the following actions can be extracted:

Read references from data store. The client is the actor of this action. References are read from the data store in the context of a client session. When the references are read from the data store, the references are converted from the storage format in which they were stored in the data store. After they have been converted, the concept of storage format doesn't exist in the client session. The client manages a list of "raw" references which do not depend on the format. There won´t be a possibility for the client to edit the references. The only thing the client can do after reading the references from the data store, is to change the storage format. This will affect the storage format to which the references in the client session will be converted upon saving to the data store. See figure 1 for a graphical representation of this idea. The client will be responsible for initiating this action

Save references to data store. When the client decides to save the references in the client session to the data store, the configured storage format is taken into consideration. The raw references are being converted to the configured storage format and are written to the data store. So references can be read from the data store in a certain storage format. Then the storage format can be changed in the client. And after, the references can be converted to the new storage format and saved to the data store. The client will be responsible for initiating this action. See figure 1 for a graphical representation of this idea.

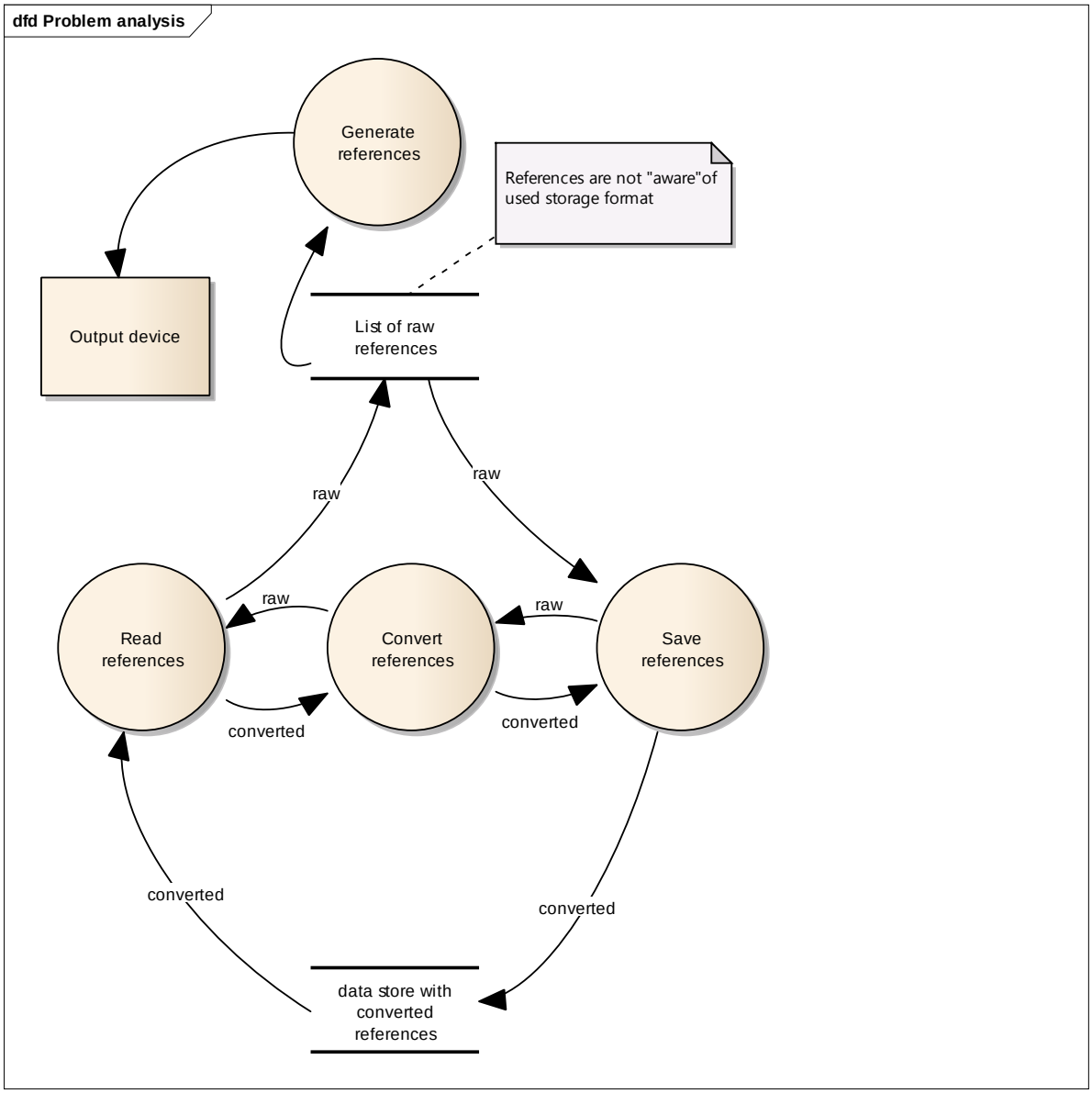

Figure 1: Problem analysis, generating, reading and saving references

Generate references. The references that exist in the client session will be generated. Before the reference can be generated, the rule check "Reference completeness" needs to be performed (see later). A reference will be responsible for generating itself. In order to determine the order of the fields, the rule check "Order of fields in the reference" will have to be performed (see later). In order to display a field within a reference, the rule check "Display format of fields in reference" will have to be performed (see later). The client will be responsible for initiating this action

Rules & strategies

This section focusses on rules and strategies that can be extracted from the case description.

Order of fields in the reference
When a reference is generated, the order of fields in the reference must be determined. The deciding factor is the reference type. The order of a field in a reference is a function of the name of the field and the reference type. The reference will know how to determine the order of its fields, as it contains the fields and it also know the type of the references. So the responsibility of this rule belongs to the reference.

Display format of fields in reference
The case description mentions that the style determines for each field how it is displayed. Every field can be displayed in a different way. This means that the display of a field is a function of the name of the field and the style. The style depends on the client session. All references within the series of use-case scenarios will have the same style (assumption, not clear from the case description). The field itself is responsible for displaying itself. It needs the help of the style that is configured in the client session. A field needs to be annotated with only one format. For example, a certain style could dictate that for the "author" field, the format "italic" should be applied. For every style, the set of fields that have a certain formatting is defined. In case no format exists for a certain field, it is assumed that the field should be displayed without any formatting, i.e. "normal" format.

Reference completeness
At the moment that a reference is generated, a precondition must be met: A check must be made that ensures that all fields that need to be present in the reference, according to the type of the reference, are present in the list of fields. The reference itself will be responsible for this rule, as the fields are contained within the reference and are the fields are not aware of each other.

Data store selection
The case description mentions that fact that references must be save to and read from a file or a database. This common denominator of these examples is the concept of data store. All references within a series of use-case interactions must use the same data store. It cannot be the case that one reference is saved to a file and another reference is saved to a database in the same composite save action. So what is the strategy of selecting the data store? The entity that is responsible for determining the data store type is the client. And this decision must be kept in the client

session during the interaction of the use-cases. The client is responsible for the selection of this strategy.

Storage format selection
The case mentions that various styles exist to display references. The client will choose the storage format and during the duration of the use-case scenarios, the selection will be stored in the client session. The client is responsible for the selection of this strategy. An extra remark needs to be made. When reading the list of converted references from the data store, the data must be inspected in order to determine what type of storage format the converted references were stored with. After the correct storage format has been noted, the correct storage formatter can be used to convert the stored references back to a list of raw references

Style selection
As stated before, the user will be able to switch between several preconfigured styles. Style can be defined in a flexible way through a configuration.

Part 2: Design

This design is divided into two parts. The first part explains how the different classes of objects relate to each other, so focusing on the "usage" aspect of the design. How certain objects are created is discussed in the other part of the paragraph.

Usage

The design can further be split into two *decoupled* subparts: The part that takes care of the generation of the references (front-end) and the part that manages the references with respect to storing and retrieving references from and to the data store (back-end).

Generation of references (front-end)

The following class diagram depicts the relationship between entities that are involved in the generation of references.

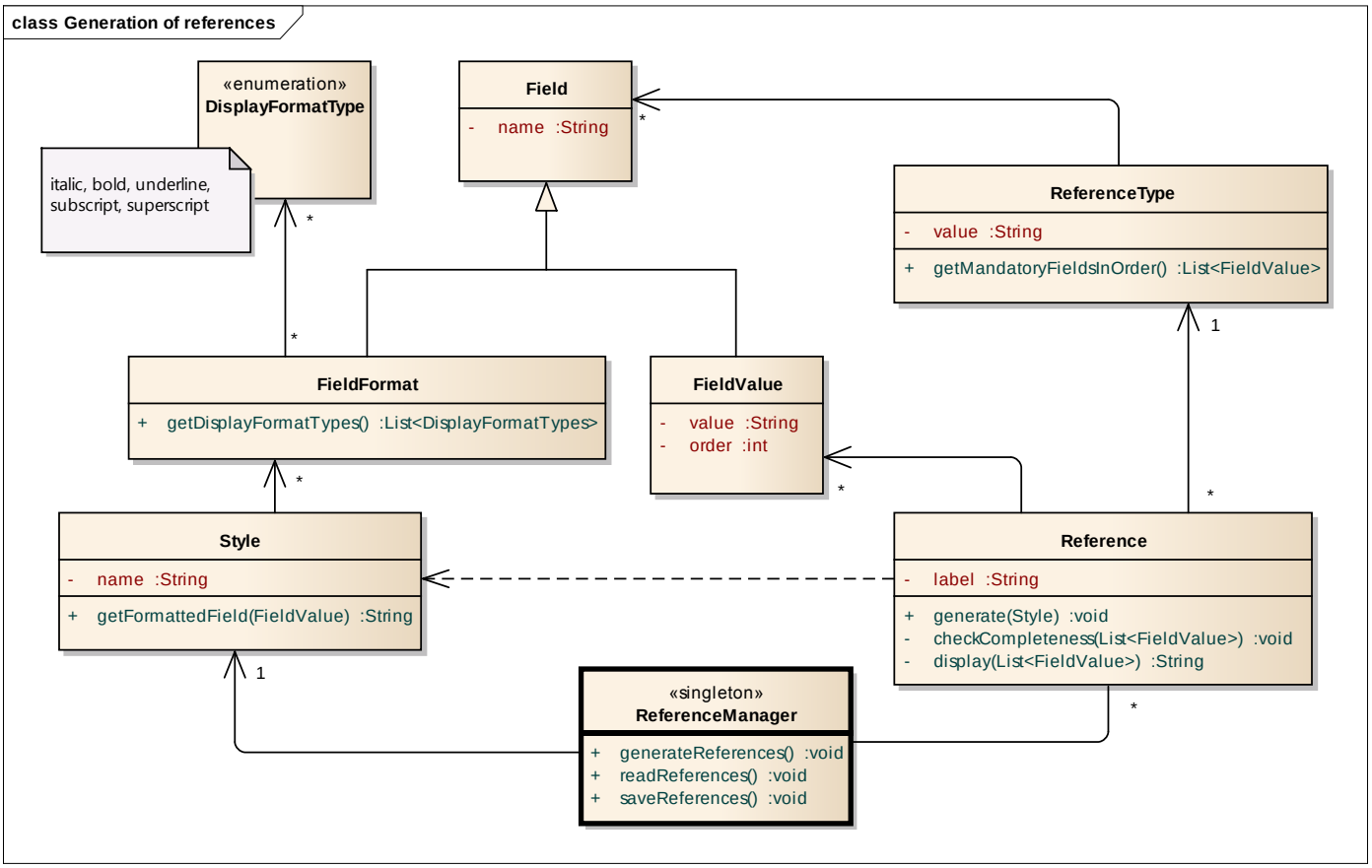

Figure 2: Class diagram of the generation part of the design

The following table contains comments that clarify the aforementioned class diagram.

Entity	Comments
ReferenceManager	This singleton represents the main application that manages the use-cases that are present in the system. When calling the method generate() on the references, it will pass the configured style to the reference object, as this object will need this information in order to correctly display the references
Reference	Every reference has a attribute of type ReferenceType and can ask this object for the fields that should be present for the associated reference type by using the method **getMandatoryFieldsInOrder**. After having identified the mandatory fields and the order in which they should be generated, the list of fields that exist in the reference is matched with the mandatory list of fields. If it doesn't match, the reference is not generated. A reference has parameter visibility on the style that is configured in the system. It will ask to style to return the formatted string representation of a certain field by calling the **getFormattedField** method on the style. This method will be called for all the fields that are available for the references and the results are concatenated and displayed
ReferenceType	An object of this type contains a list of mandatory objects of type **FieldValue**, which are maintained in a sorted order according to the order in which fields should be displayed during the generation of the reference
FieldValue	This class inherits the name of the field from **Field** and adds a value and an order to it
Field	Class that defines the name of a field
FieldFormat	This class inherits the name of the field from **Field** and adds a collection of objects of **DisplayFormatType**s to it that defines how the field should be formatted. A field can be annotated by multiple format types.
Style	An object of type Style will contain a list of objects of type **FieldFormat**. Style can be configured and the **ReferenceManager** can choose an appropriate instance of type **Style** which should be applied to all the references in the context

Table 1: Clarification of generation class diagram

The figure below shows how references, fields and the reference type relate to each other in a examplaric way.

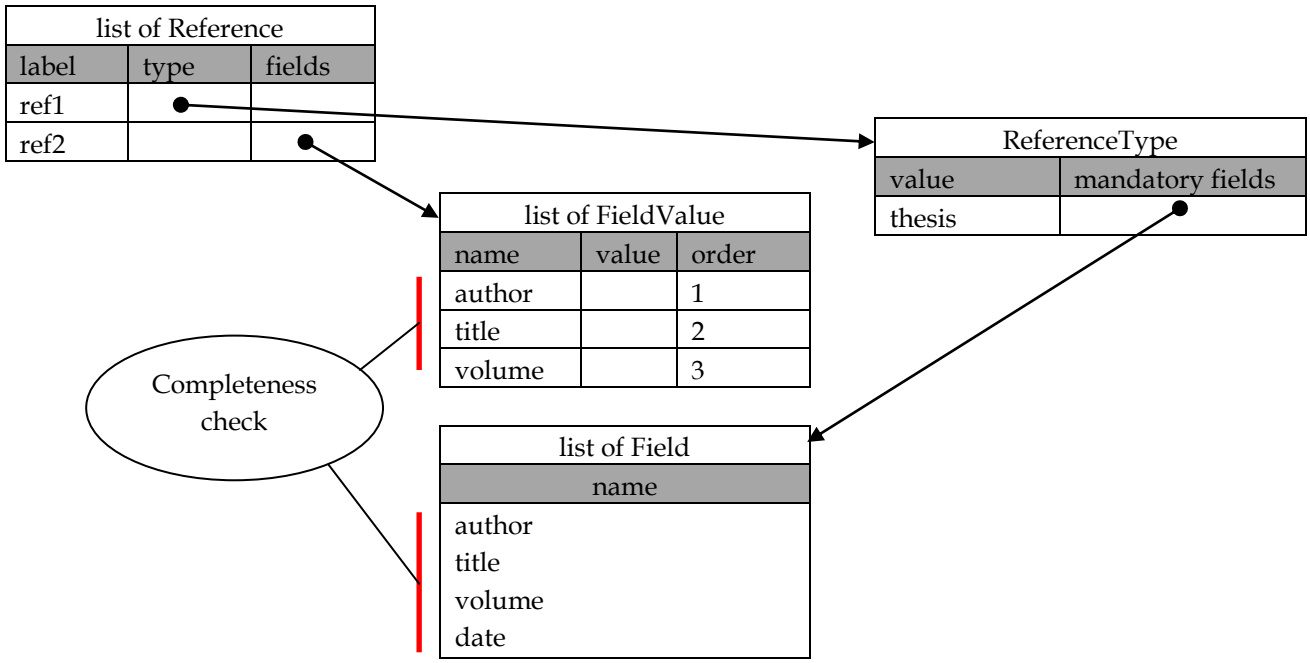

Figure 3: References, ReferenceTypes, Fields and FieldValues

Styles, field formats and display format types are explained in the following figure:

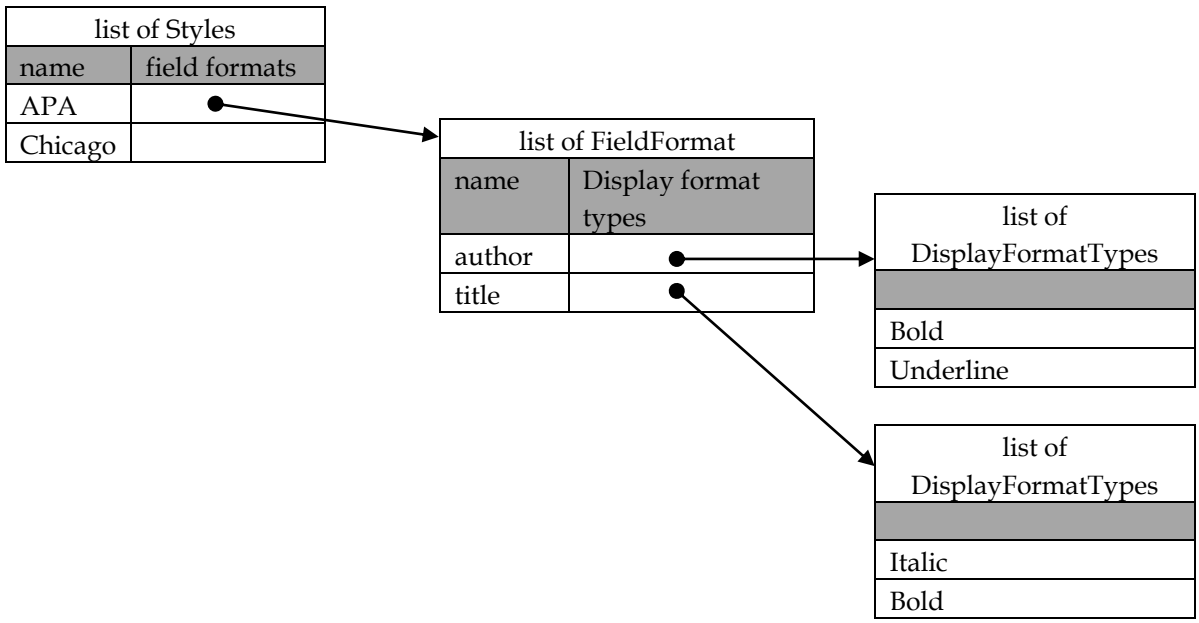

Figure 4: Styles, FieldFormats and DisplayFormatTypes

The interaction of messages between objects is depicted in the following sequence diagram.

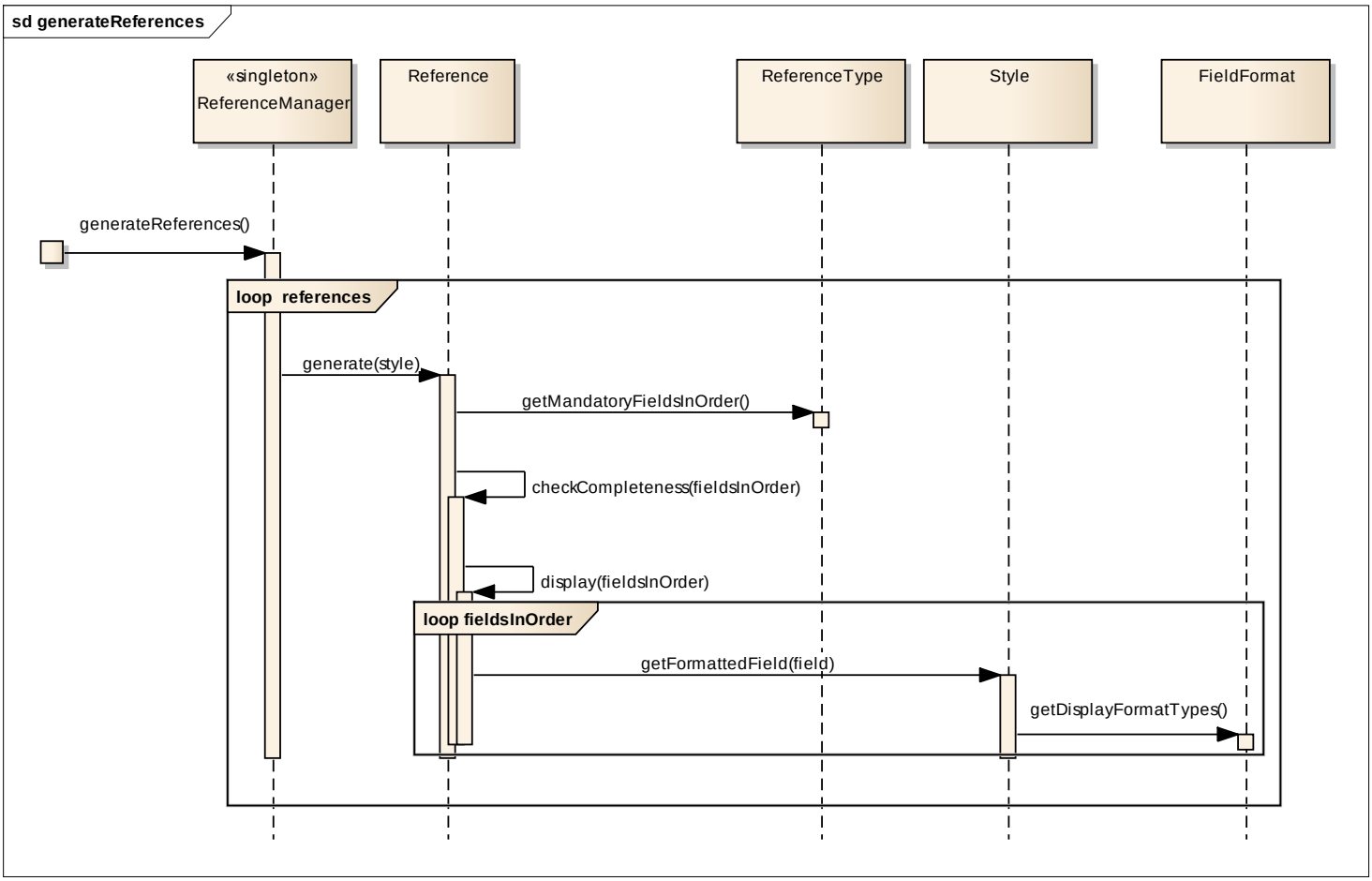

Figure 5: Sequence diagram to illustrate the interaction of objects during the generation of references

As said before, the front-end (generation of references) and back-end (reading and saving of references) has been decoupled. The front-end is not aware of the internal storage format that is used to store the references in the data store. The type of data store and the type of storage formatter can be changed without impacting the front-end. The interface between the front-end is accomplished by the usage of a list of "raw" references (references that have no notion of the storage format) that is passed from the front-end to the back-end when calling the save functionality and vice-versa in the case of retrieving the references from the data store.

Reading and saving references (back-end)

The following class diagram depicts the relationship between entities that are involved in the I/O part of the managing of the references.

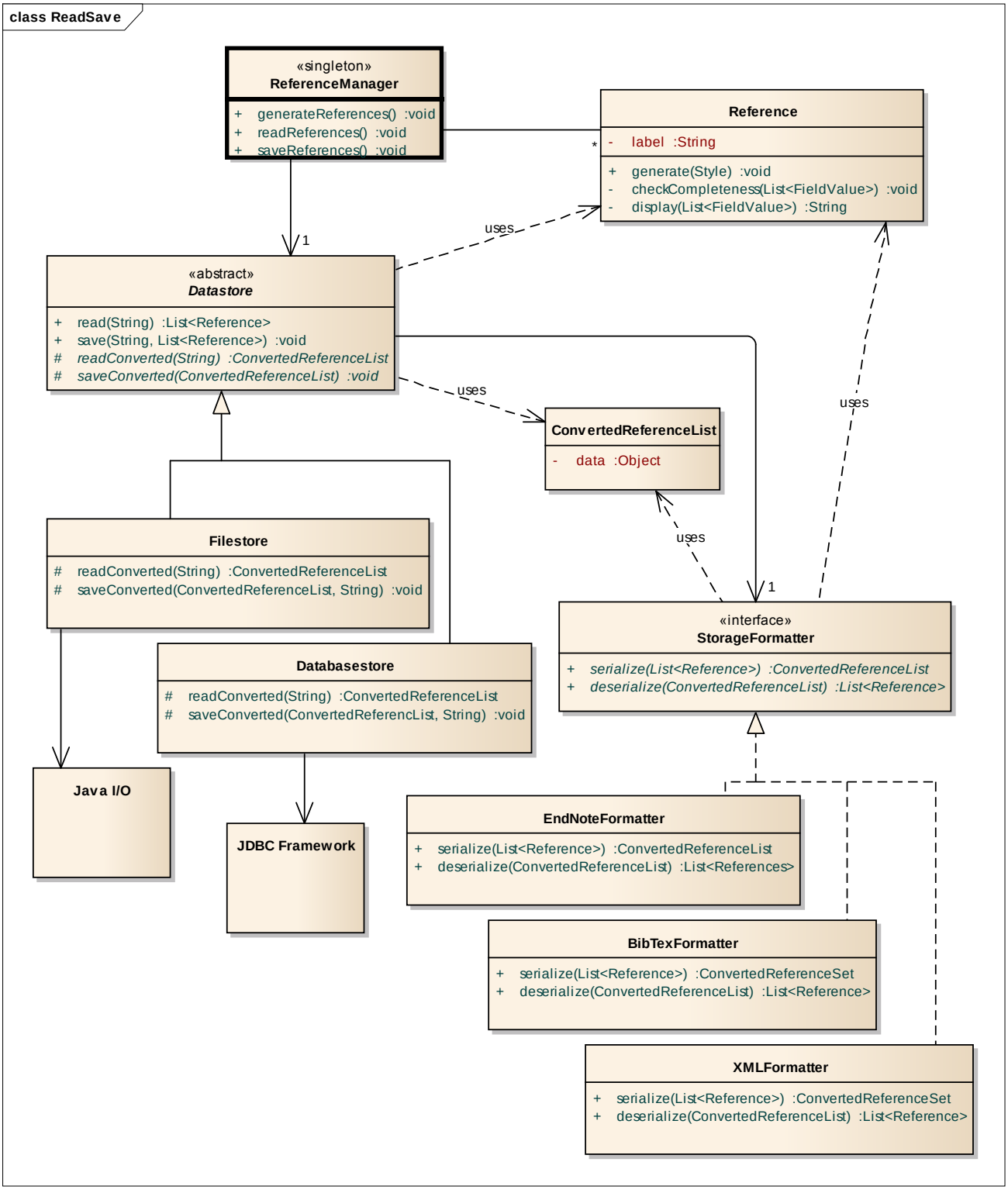

Figure 6: Class diagram for the I/O part that manages references

The following table contains comments that clarify the aforementioned class diagram.

Entity	Comments
ReferenceManager	The controller class that was already managed before. It uses read and save commands to the configured data store and serves as a linking pin between the front-end and the back-end
Datastore	Abstract class that *knows how to save and read a list of raw references to and from a certain implementation*. Currently, three implementations are modelled, but extra implementations could be added with minimal costs. An object of this class has a reference to the used **StorageFormat**. StorageFormat is an interface, so the datastore has no knowledge of what kind of StorageFormat is being used
Filestore	Concrete implementation of a **Datastore**. It also serves as an adapter because it abstracts the logic of communicating directly with the Java I/O system
Databasestore	Concrete implementation of a **Datastore**. It also serves as an adapter because it abstracts the logic of communicating directly with the JDBC framework
StorageFormatter	An interface that establishes a contract of the serialization (raw to converted) and deserialization (converted to raw) of references. During serialization, a list of raw references is converted to an object of type ConvertedReferenceList which contains the information in a low-level format. During deserialization, this information in low-level format is converted back to the list of raw references
EndNoteFormatter	Concrete implementation of a **StorageFormatter**. It also serves as an adapter because it abstracts the logic of serialization (raw to converted) and deserialization of references in EndNote format
BibTexFormatter	Concrete implementation of a **StorageFormatter**. It also serves as an adapter because it abstracts the logic of serialization (raw to converted) and deserialization of references in BibTex format
XMLFormatter	Concrete implementation of a **StorageFormatter**. It also serves as an adapter because it abstracts the logic of serialization (raw to converted) and deserialization of references in XML format
ConvertedReferenceList	Class that represents a serialized list of references which is used to be saved to a datastore
Reference	Class that represents a raw references that is used to generate references

Table 2: Clarification of I/O class diagram

The following sequence diagram illustrates the interaction between messages that are involved in the *reading of references* from a data store.

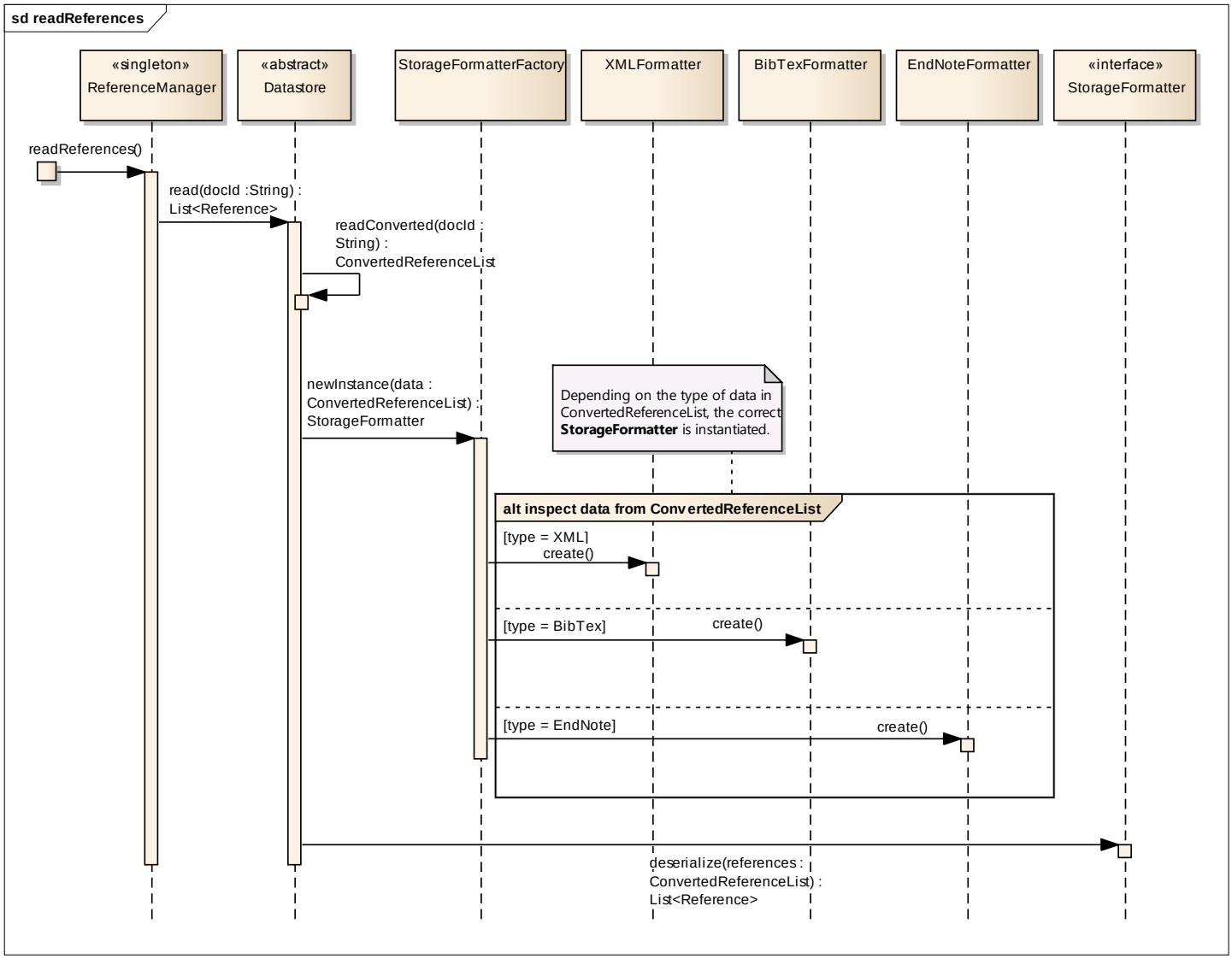

Figure 7: Interaction of messages when reading references from a datastore

Comments:

1. The interaction starts when the ReferenceManager fires the **readReferences** method. This method is aware of the context of the document, i.e. the document id that is being used at that moment
2. It passes the document id to the read method of the reference that identifies an object of an implementation of Datastore class. This implementation retrieves the converted references
3. A call is made to an object of type **StorageFormatterFactory** and the retrieved data (**ConvertedReferenceList**) is passed as a parameter. StorageFormatterFactory inspects that data and

knows which **StorageFormatter** to instantiate. This instance is returned to the datastore object

4. The datastore object calls the **deserialize** method on the object of type StorageFormatter (*which* one, the datastore object is not aware of). The StorageFormatter converts the object of type ConvertedReferenceList back to a list of raw references, which in turn are returned to the ReferenceManager

The saving of references is done in a similar way, except for the fact when converted references are being saved, it's not necessary to detect the correct StorageFormatter, because this logic is a user-initiated action. Before a user adds references to the document, he/she will set the appropriate data store and storage formatter. At that time, the correct instances of both types are being created. When references are saved to the data store, these settings are used subsequently.

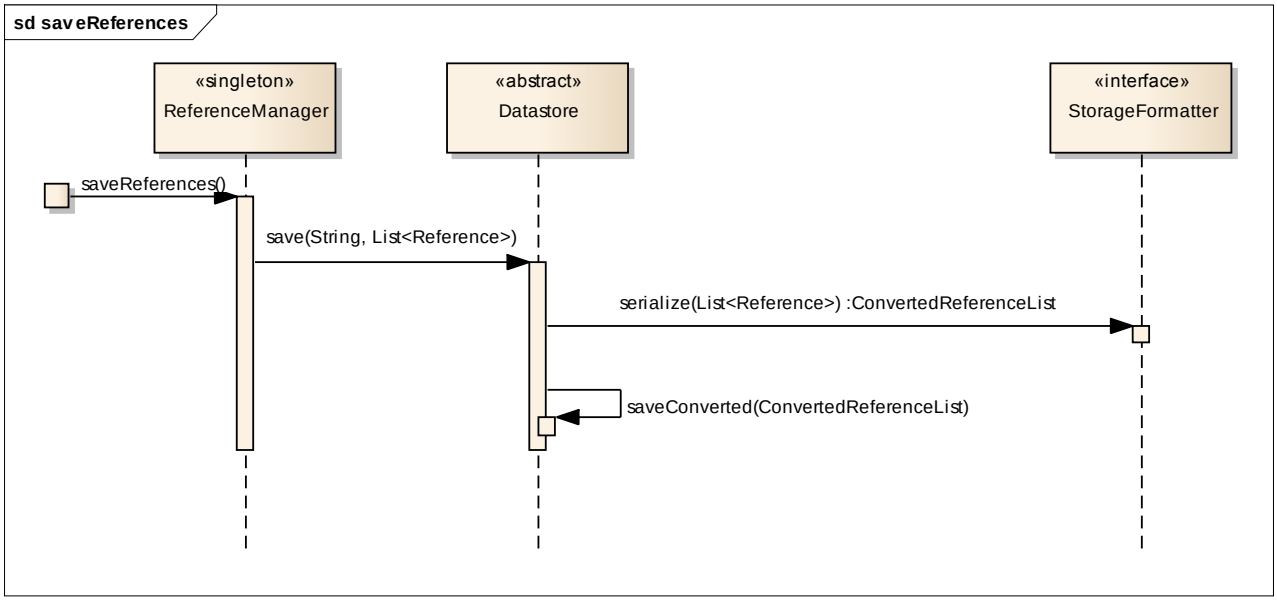

Figure 8: Interaction of messages when saving references to a datastore

Creational aspects

In the previous section, the relationship between classes was discussed in the context of the usage. This section will focus on the creational part of the design: Which objects will be responsible for the creation of other objects?

The next figure depicts the class diagram that contains classes that are responsible for the creation of other objects.

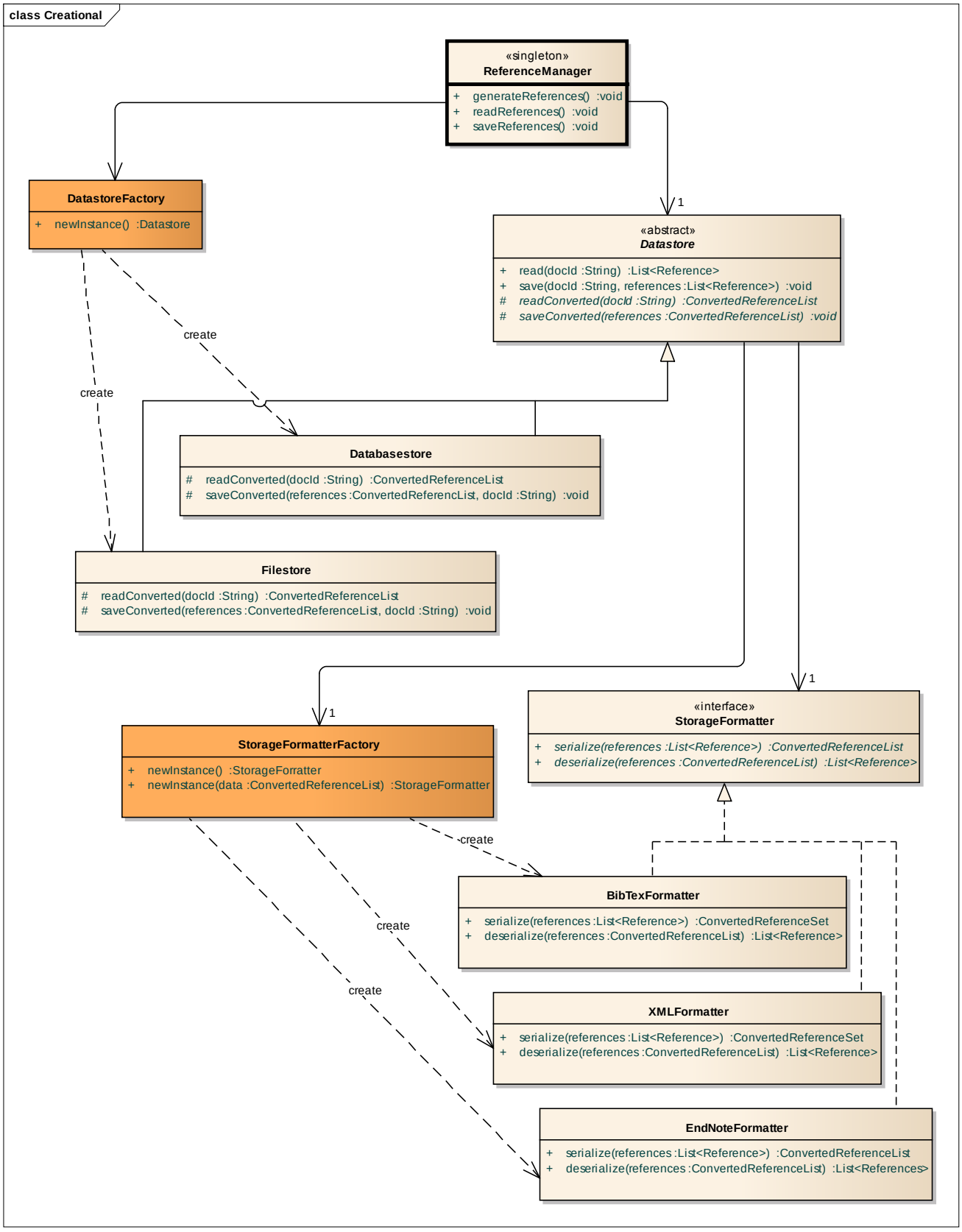

Figure 9: Class diagram that mentiones classes with a creational character

The singleton class ReferenceManager has an attribute that is a reference to an instance of a Datastore. At startup of the application (by configuration), or when the user decides to switch the data store, the ReferenceManager makes the **DatastoreFactory** responsible for creating the appropriate instance of a Datastore.

An example code snippet of this DatastoreFactory looks like this:

class code

```
package nl.apg.common.annotation;

public class DatastoreFactory {
    Configuration configuration;

    public DatastoreFactory(Configuration configuration) {
        this.configuration = configuration;
    }

    public Datastore newInstance() {
        if (this.configuration.equals(Configuration.Store.DATABASE)) {
            return new Databasestore();
        }
        else if (this.configuration.equals(Configuration.Store.FILE)) {
            return new Filestore();
        }
        else {
            throw new UnimplementedException("No options available for " + this.configuration);
        }
    }
}
```

Figure 10: Example code snippet that explains the workings of the DatastoreFactory class

After an instance of a Datastore class has been created, the Datastore instance itself needs to be provided with an instance of the StorageFormatter class. The interaction of objects that are involved has already been discussed in *Figure 7: Interaction of messages when reading references from a datastore* in the context of the reading of the references from the datastore. In a similar fashion, the Datastore instance is given the correct instance of the StorageFormatter class.

So, the *ReferenceManager* delegates the task of the creation of *Datastore* instances to the *DatastoreFactory*. After the *Datastore* instance has been created (and assigned to the ReferenceManager), the *Datastore* instance delegates the task of the creation of the *StorageFormatter* instance to the *StorageFormatterFactory*.

Complete class diagram

Thte complete class diagram can be found on the next page.

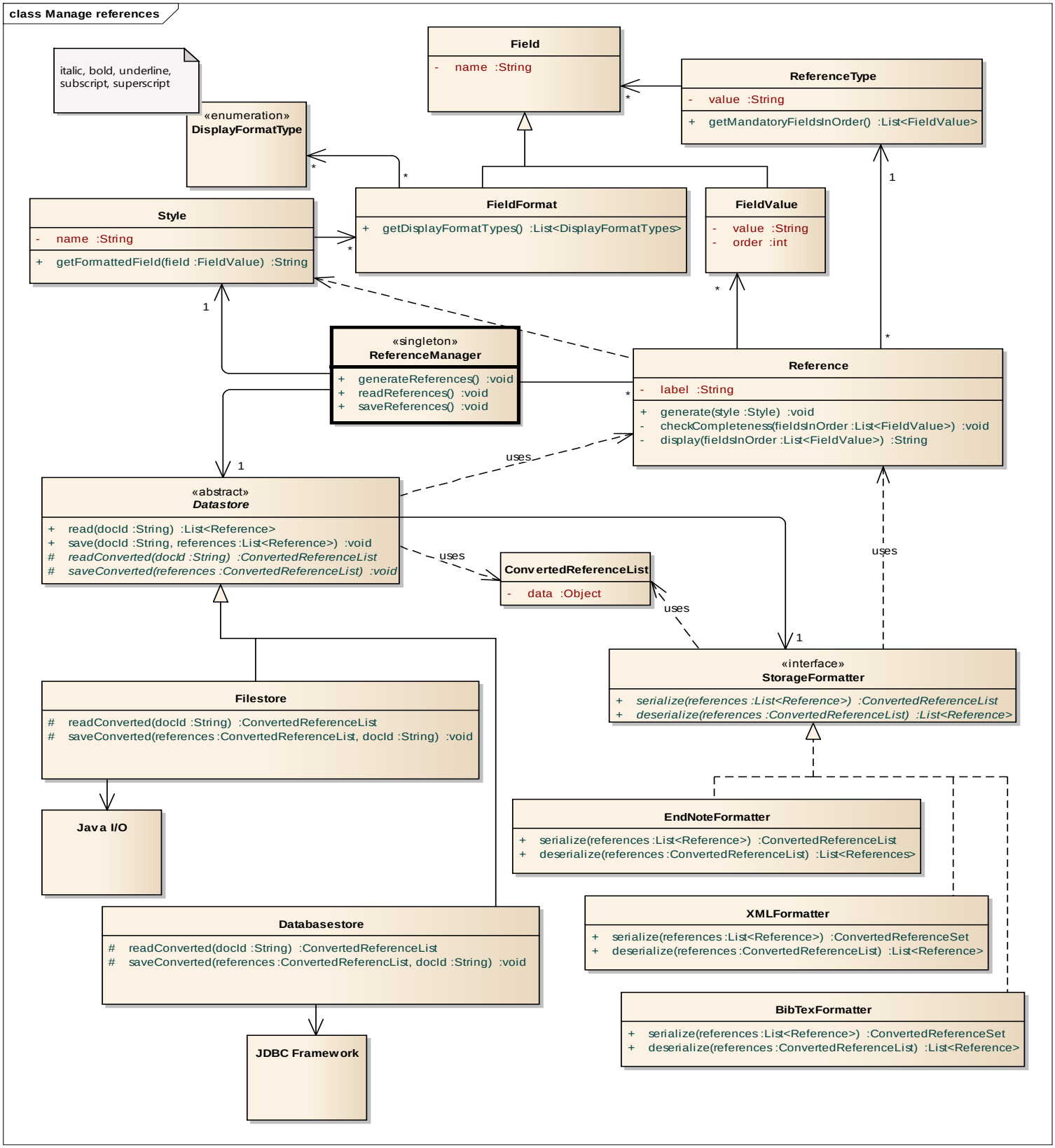

Figure 11: Complete class diagram

Part 3: Patterns

This chapter discusses the design patterns that were used in the design.

Bridge Pattern?

I've used a pattern that looks like a bridge in order to solve the problem of two varying concepts that should be loosely coupled. But the semantics are a bit different. This leads to an interesting discussion that I would like to introduce in this section (and elaborate more on in the next section "Design decisions"). The following diagram shows the involved classes.

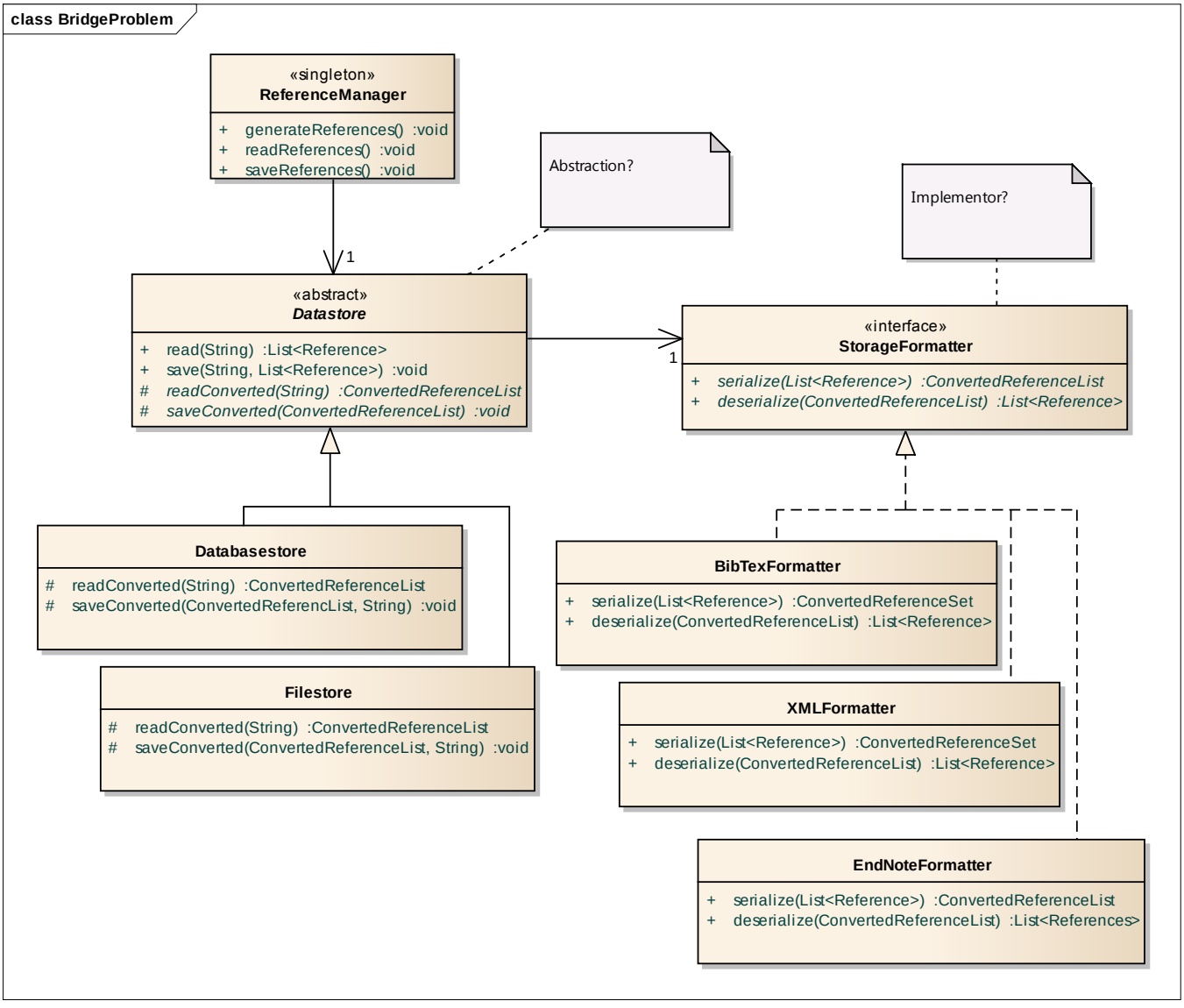

Figure 12: Decoupling ReferenceManager, Datastore and StorageFormatter… but is it a bridge?

The two varying aspects are the Datastore and the StorageFormatter. As can clearly be derived from the class diagram, ReferenceManager, Datastore and StorageFormatter are loosely coupled. Datastore and

StorageFormatter implementations can be added without impacting the design. The most compelling reason to use a Bridge Pattern according to GoF is to avoid a permanent binding between an abstraction and its implementation.

In the textbook, an example was given about Shapes (abstraction) and its implementation (Drawing). Shapes can have several concrete forms and how shapes should be drawn is done by the Implementor, the Drawing. Drawings can be implemented in several ways.

But in the design that I've given, there is no such "Abstraction-Implementation" relationship between Datastore and StorageFormatter. A Datastore is not implemented by the StorageFormatter. A StorageFormatter is merely used by the Datastore, but that's how for it goes.

So, is it a Bridge Pattern? No. it's not. There are 'just' two Strategy Patterns involved, which will be explained in the next section.

Two cases of Strategy Patterns

There is a need to use different functionalities regarding the storage of the references according to the use case description. References can be stored either in a file or in a database, based on a configuration setting or on an overruling user initiated action (assumed).

The solution to this functionality is the usage of a Strategy Pattern (citing the book):

> Solution: Separates the selection of algorithm from the implementation of the algorithm. Allows for the selection to be made based upon selection

The **ConcreteStrategies** are the Filestore and the Databasestore. The role of **Context** is played by ReferenceManager who maintains a reference to the **Strategy** (Datastore). The ReferenceManager forwards request to the datastore instance.

The Strategy Pattern was also used to decouple the Datastore and the StorageFormatter. There are three **ConcreteStrategies** of the **Strategy** StorageFormatter (EndNoteFormatter, XMLFormatter and BibTexFormatter). The **Context** in this case is the abstract class Datastore. The algorithm that is being abstracted is the serialization and deserialization of references.

Two cases of Factory Patterns

The Factory Pattern is used twice (Not to be confused with the Abstract Factory Pattern). The next chapter will explain about the decision to use this pattern instead of the Abstract Factory Pattern or the Factory Method Pattern.

There are two Factories, a Datastore Factory and a StorageFormatterFactory.

A DatastoreFactory has logic in place to select from two available **ConcreteProduct**s: Databasestore and Filestore. Both of these concrete products inherit from the **Product** Datastore.

StorageFormatterFactory has logic in place to select from three available **ConcreteProduct**s: BibTexFormatter, XMLFormatter and EndNoteFormatter. These three concrete products inherit from the **Product** StorageFormatter.

Two cases of Adapter Patterns

The Adapter Pattern is used twice.

The Filestore and Databasestore implement two abstract methods that are defined in the Abstract class Datastore: readConverted and saveConverted.

In the case of Filestore, readConverted and saveConverted must interface with a low-level API to open and close files and to read and write byte streams to Operating System file structures. In this design, the Java I/O system must be abstracted from the application itself. So a unified interface to a set of interfaces in the Java I/O subsystem must be provided. The readConverted and saveConverted of Datastore serve as interfaces that abstract Java specific logic.

In case of Databasestore, readConverted and saveConverted must interface with a low-level JDBC calls to query the database and insert rows in the database. The JDBC functionality can be treated as a separate subsystem and it is desired that these fine grained interfaces are abstracted into a unified interface that is provided by Databasestore.

These two examples are examples of Adapter Patterns, not Façade Patterns. A Façade Patterns simplifies an interface. In this case, an existing interface is being converted into another interface.

Singleton Patterns

I used one singleton, the ReferenceManager. This is the controller of the application and only instance will be available of this class. When implementing this class, a special method will be called (or static block will be activated) that will check whether the object has already been instantiated before.

Part 4: Design decisions

This chapter focusses on decisions that I made when making the design.

Modelling references, formatters and data stores

The first decision is regarding how to model references, storages formats and data stores in relation to design patterns.

In the design that I proposed I investigated how existing framework implement the persistence and retrieval of objects. I looked at the API of Hibernate, an ORM framework for mapping objects in Java/.NET to rows in a relational database. Although this is a technical implementation issue, there are design patterns involved which also find its reflection in many designs that are related with persistence and retrieval of objects.

In this framework, there is the concept of Session which takes care of preparing statement, transaction management, executing queries/DML statements and etcetera.

A Session provided methods:

- to prepare a statement based on a query string
- to execute a prepared statement
- insert/update/delete/ retrieve objects

So in order for objects to be retrieved, inserted, deleted and updates, responsibility of these tasks are referred to this Session object and the objects are passed as parameters.

This is a very intuitive way of interacting with a data store and I instinctively headed for that direction when designing the relationships between references, storage format and data stores: A data store will receive a reference (or list of), ask a StorageFormatter to convert it to the desired format and then give the data store the responsibility to store the converted reference.

After the designing the classes and relationships, I ended up with the design that is depicted in Figure 12. I decoupled two varying and related concepts, i.e. Datastore and StorageFormatter. Of course I recognized the

Bridge Pattern, but the semantics were not correct as the **Datastore** is not abstraction of the **StorageFormatter** implementation. So there is no Bridge Pattern, these are *just* two linked Strategy Patterns. I thought: What a pity! Immediately I started thinking of how to redesign it so a nice Bridge Pattern would appear. The following class diagram appeared.

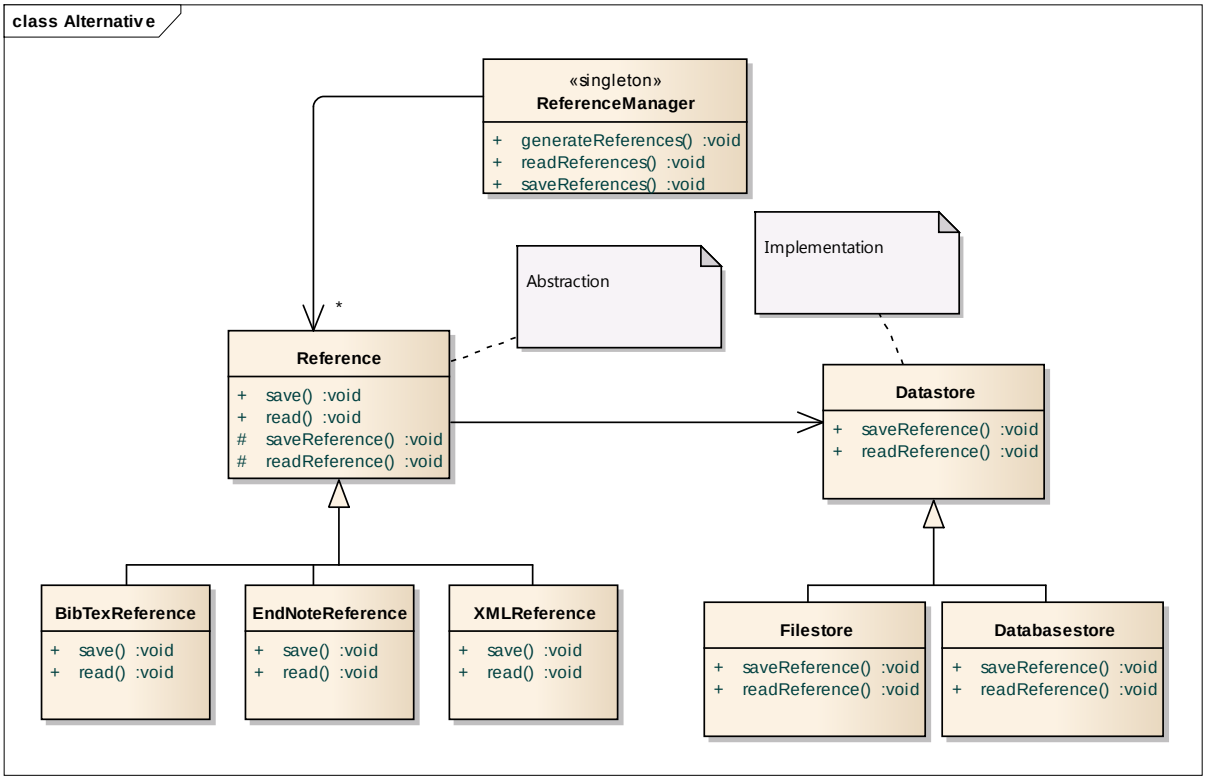

Figure 13: An alternative to modelling references, data stores and formats

A reference is an abstraction with several refined abstractions which use classes derived from Datastore (Implementor), without knowing which particular ConcreteImplementor is in use. But there is a problem! Back to Hibernate.

An object does not have an interface to insert/update/delete or retrieve itself, as these objects are plain simple objects, which are defined out of the context of Hibernate.

In order for a business object to retrieve itself from a data store, it would first have to create an empty object (who will do this?), set the identifier to look for in the data store (who will do this?), then delegate to the Session object to perform the operation. This is not intuitive way of communicating between the Session object and the business objects and it's also not possible.

So these are my thoughts on how I came up with this part of the design that I proposed in the previous chapter (Figure 12). Perhaps it is a Bridge Pattern after all, and perhaps I am just confused by the nomenclature, semantics that is used in the description of the Bridge Pattern on page 185 of the textbook and also in the description of the pattern by GoF.

Factories

The factories StorageFormatterFactory and DatastoreFactory are designed using the Factory Pattern. It's not a pattern that exists in the list of patterns that are provided on the Design Patterns CD, but it is pattern that I've seen in other books and online resources (https://en.wikipedia.org/wiki/Factory_(object-oriented_programming, http://www.oodesign.com/factory-pattern.html). It's a simplification of the Abstract Factory Pattern that suffices for the creation of objects in the ReferenceManager design. The Abstract Factory Pattern is useful if families of related objects need to be instantiated. In the example in the textbook there is a set of LowRes objects and a set of HighRes objects that should be available for both PrintDriver and DisplayDriver. This analogy is not present in the ReferenceManager case, and to model it using an Abstract Factory Pattern would be too much in my opinion. I also didn't foresee any extensions of the case that would justify the usage of an Abstract Factory Pattern.

Part 5: Future changes

The following changes can be made with minimal or no impact on the design:

Styles and reference types

New styles and reference types can be added without any impact to the design of the application.

Display format

The use case mentions that the configuration of display formats is mutually exclusive. This means that for a certain style, a field can be displayed italic, or underlined, or bold or without any formatting. The current design makes it possible for a field to be decorated with multiple formats (one field can be both italic, underline and bold for example). In case the requirements change, this does not impact the design of the application.

Data store

In case a new data store is introduced, let's say a web service, the impact on the design will be minimal. The following activities will have to be performed in that case:

1. A new **Webservicestore** class that extends Datastore will have to be added to the design
2. The DatastoreFactory will have to be extended with an extra branch that enables for the instantiation of the newly added implementation of the Datastore

Storage formatter

In case a storage format is introduced, let's say JabRef, the impact on the design will be minimal. The following activities will have to be performed in that case:

1. A new **JabRefFormatter** class that implements StorageFormatter will have to be added and the methods for serialization and deserialization will have to be implemented
2. The StorageFormatterFactory will have to be extended with an extra branch that enables the instantiation of the newly added implementation of the StorageFormatter