**Table of Contents**

[Library Database Management System Design 3](#_Toc103529449)

[Overview 3](#_Toc103529450)

[Model: Datapoint 3](#_Toc103529451)

[Book & Student Datapoint 3](#_Toc103529452)

[Borrow & Return Datapoint 3](#_Toc103529453)

[Datapoint Enums: Type, Edge, and Book Genre 4](#_Toc103529454)

[Model: Dataset as a Datapoint Collection 4](#_Toc103529455)

[View: Menu 4](#_Toc103529456)

[Controller: Library System 5](#_Toc103529457)

[Library Database Supporting Modules 5](#_Toc103529458)

[Index Interface & Index Collection 5](#_Toc103529459)

[Datapoint Creation: Request Interface & Abstract Factory Chain 6](#_Toc103529460)

[Persistence: Serialization & JSON Export 6](#_Toc103529461)

[Avenues for Development 6](#_Toc103529462)

[Recommendation System 6](#_Toc103529463)

[Book Genre Enum: Inventory Maintenance & Searching 7](#_Toc103529464)

[Summary 7](#_Toc103529465)

[References 7](#_Toc103529466)

[Figure Appendices 8](#_Toc103529467)

[Figure 1: System Design 8](#_Toc103529468)

[Figure 2: Library System 8](#_Toc103529469)

[Figure 3: Datapoint Interface Hierarchy 9](#_Toc103529470)

[Figure 4: Index Entry Interface Hierarchy 9](#_Toc103529471)

[Figure 5: Abstract Index Hierarchy 10](#_Toc103529472)

[Figure 6: Principle of the Index Interface 10](#_Toc103529473)

[Figure 7: Principle of the Index Collection 11](#_Toc103529474)

[Figure 8: Abstract Dataset Chain Hierarchy 11](#_Toc103529475)

[Figure 9: Request Interface Hierarchy 12](#_Toc103529476)

[Figure 10: Abstract Factory Chain 12](#_Toc103529477)

[Figure 11: Datapoint Construction 13](#_Toc103529478)

[Figure 12: Example JSON Export 14](#_Toc103529479)

[Figure 13: Library Database Management System Class Diagram 15](#_Toc103529480)

[Table Appendices 16](#_Toc103529481)

[Table 1: Core Modules 16](#_Toc103529482)

[Table 2: Enums Supporting System 16](#_Toc103529483)

[Table 3: Indexed Datapoint Attributes 17](#_Toc103529484)

Library Database Management System Design

Overview

As shown in Fig-1, a customized MVC pattern was used to design the library database management system (LDBMS). The design allowed for developing each component separately before focusing on tying them together for a prototype to meet the requirements of the brief for the system, although the preference here was “bottom-up” following initial planning (Sciore, 2019). In brief, the menu module (i.e view) is exposed to the user allowing them to perform create, read, update & delete (CRUD) operations via the library system module (ie controller) in Fig-2. The singleton controller uses a dataset chain module to serve as a database for the underlining dataset module and an abstract factory chain for the construction of valid data points (ie the Book, Student, Borrow and Return models) in Fig-3.

Model: Datapoint

Book & Student Datapoint

The design of the datapoint interface module in Fig-3 originated during the initial planning phase. Where it was viewed that all underlining data points (ie Book, Student, Borrow, Return) were largely declarative in their function because their destination was in a database. Meaning, that requests for interactions between data points are done by queries from user interaction (e.g searching/getting a specific student or book) and not the data points themselves. These interactions are supported by specific modules, such as a custom first in first out queue attribute on books, which allows a student ID to be added to a book if it is already borrowed. Or a custom set attribute on students to restrict the amount books that they can borrow, while also working towards functionality to allow students to borrow multiple books but not the entire library, and so the size of the underlining array is limited by a constants module with a current value of 2. While both data points also store the number of borrow & returns events they were involved in, the design was further developed for students to store the number of genres that they have borrowed with an eye toward a recommendation system further discussed in “*Avenues for Development*”. From the broader view of these two data points, they were grouped into instances of an abstract item in the library (e.g books as stock/inventory and students as customers).

Borrow & Return Datapoint

Borrows & returns were viewed as being declarative interactions between these items, and so were then grouped into an abstract activity module. The abstract activity model declares the date on which interaction between two items occurred, their expected end/return date, and whether or not the activity was resolved on time or was late (eg return state is stored by the system). Additionally, modeling borrow/returns in this light also lead to viewing the borrows dataset as being dynamic, and returns as expanding because all borrows should resolve into a return given that they involve the same items. Furthermore, storing both does not offer additional valuable information, and so a return gets constructed from borrow, and if successful that borrow is deleted.

Datapoint Enums: Type, Edge, and Book Genre

With the established hierarchy of the datapoint interface, enumerated types were then designed so that the datapoint interface can be strategically used in conjunction with other design patterns to better maintain single responsibility during development (Table 1-2). A “Datapoint\_Type” enum was used to facilitate operations over the abstraction level of datapoints (ie Item or Activity), which was used by the systems abstract factory chain of command in conjunction with the request interface to construct data points (Fig 9-10). A “Datapoint\_Edge” was developed for operations over specific types of data points, such as object construction, querying datasets, and index creation. Finally, a Genre enum was developed to incorporate some element of quality control on importing, with an eye for future functionality “*Avenues for Development*”. It was developed from noticing in the test import data that there were different ways inventory with no actual genre data was represented, and also allow functionality to be added to help the user maintain/categorize their inventory.

Model: Dataset as a Datapoint Collection

Since one of the requirements of the brief was to avoid the use of database services such as MySQL, a custom database module was designed (Fig-8). The database was designed using an adapted chain of command pattern so that each concrete dataset can accept or delegate interactions with its associated list of data points and indexes (Sciore, 2019). While the design facilitates treating a list of data points as a table, and a collection of datasets as a database, it creates an overhead of doubling methods, questionably achieves “single responsibility” and is difficult to maintain. Where public methods are used to accept/delegate a method on a specific dataset, then private or abstract methods are used to run the associated method. While functional, in retrospect a design similar to the index collection would be better suited to this specific instance (the dataset chain was developed before indexing). Where in short, the attribute of this “Database” object would be a hashmap, where the keys are the valid “Datapoint\_Edge” enum and the values are the corresponding datasets. The methods that are exposed to the client would then be effectively similar to the current chain, only the Datapoint\_Edge arguments are used to get the appropriate dataset instead of delegating/accepting a method call. Finally, to support searching/sorting operations an Index Interface was developed and is discussed in the “*Index Interface & Index Collection*” section (Fig 4-7).

View: Menu

The menu is the client-facing module where a user receives prompts to perform CRUD operations on the LDBMS (Fig-1). The menu client composes the menu interface, allowing for future login-related functionality to be added, and the menu module itself instantiates the library system (Fig-2). An IO-Utilis module (not discussed in this report) guides the user through valid interactions with the LDBMS. The presentation of the menu was supported by the development of a “Boundaries” enum which served to standardize the applications title screen, sectioning between queries and results along with headers and footers for these sections, where the rationale for its development was toward user experience. The menu also imports most enums to facilitate the user’s creation of valid objects and querying valid indexes from valid datasets, which lowered the need for creating custom exceptions. Persistence of the LDBMS occurs automatically on start-up and shutdown, which calls the appropriate (de)serialization of each data point for each dataset.

Controller: Library System

The library system module was designed using a singleton pattern so that only one instance can exist (Fig-2). The library system is initialized with a null dataset chain serving as the database, and an abstract factory chain allowing for concrete datapoint creation via the request interface. During its initial construction, each dataset is deserialized into a list of data points and then imported into the database. The importing process first sets the dataset property to the input list, and through the index collection module, all indexes are created simultaneously from one iteration of the input list (not a concurrent process). The rationale for designing the construction this way was towards minimizing the Big-O complexity of constructing the LDBMS but also to simplify interactions with the datasets and their indexes (Fig 7-8)

Library Database Supporting Modules

Index Interface & Index Collection

By itself, an index is a list of index entries structured as per (Fig 4-6). Where in short, an index entry has a property that represents a sortable/searchable value and another property that points to its position in an external array. An index is then created from a list of these index entries, which uses their comparison methods for implementing sorting (eg bubble & merge) and searching algorithms (eg binary & linear). Such as using binary search to find a target or nearest value, then linear search to extend this result to a range of indexes satisfying the search criteria. Currently, the complete indexing module supports strings and integers which has enabled configuring the LDBMS with the indexes shown in table 3. Furthermore, the indexing scheme enables the continual use of efficient searching algorithms (ie binary searching) because each index can be sorted independently. This allows for the 0-indexed auto-incremented datapoint IDs to be used as a primary key that points to a datapoint’s row number in the dataset. The design of the indexing module also facilitates the development of new searching/sorting algorithms without having to re-design other modules of the LDBMS, outside of exposing these new methods. Although not currently implemented in the menu, examples here include finding the nearest integer, finding the nearest integers in a range, and searching strings starting with the input query. However, in its current form, the module can be simplified from the use of enums to support the various specific searching & comparison types.

Since a data point has a collection of possible attributes that can be indexed, a separate “IndexCollection” module was designed to provide a dataset with a single object to best utilize the designed indexing strategy (Fig 7-8). Where the hash map of valid dataset indexes is configured during construction via the “Dataset\_Index” (Table 3). The “Dataset\_Index” is used as a key for exposing the dataset with methods for utilizing its string and integer indexes. A limitation of the current design is that new data created by the user is only added to the dataset and not the index entry list. While the LDBMS does have methods for reconstructing the index collection and is provided to the user, its current implementation is limited. In short, this is because it was speculated that a given user session may not generate more data than is in already in the system, meaning updating the indexes could be handled by using the provided method/restarting the system naturally. Given this maintaining deletions, insertions & resolutions through the initial design plan for a separate unsorted index entry list that is searched linearly, after the sorted index entries are binary searched was not implemented. Nor was exploring a binary search tree index with a custom iterator for breadth/depth-first traversal.

Datapoint Creation: Request Interface & Abstract Factory Chain

The rationale for the request interface was to enable the “Datapoint\_Edge” enum to be able to create their associated data point and allow the abstract factory to create a datapoint edge from a specific request, as shown in Fig-9 (Sciore, 2019). The solution overcame the challenge associated with different constructors of the different datapoint edges using a hash map of strings. Where the key for each element in the set is an attribute for the constructor, and its value is the value that the datapoint should have. The request interface also simplified passing requests for item creation between the view and the model. Where currently an abstract factory chain is responsible for passing these requests for the construction of specific datapoint edges to the Datapoint\_Edge enum (Fig 10-11). There is a concrete factory for each datapoint type (ie Item and Activity) that is responsible for constructing the associated edges such as Book & Student for the Item datapoints, then Borrow & Return for the Activity datapoints.

Persistence: Serialization & JSON Export

The persistence of the system was achieved by implementing the serializable interface on the abstract data point level and any class these objects store, such as the custom student ID queue & book ID borrow set. Serialization of the data points occurs automatically when the user requests to quit the system on the menu, where the list of concrete data points associated with a specific dataset is iteratively written to/read from a byte stream. Whereas (de)serialization occurs during startup.

While not updated since the development of the custom student queue & book set, the system does permit exporting datasets in JSON format. The rationale was that these data could be analyzed by another system such as R or Python-Scipy (Fig 11-12). Additionally, while the Item datasets are importable as JSON, the Activity datasets are not because the construction of activity objects requires a book & and a student object. A process that creates more & needless overhead during the construction of the LDBMS, compared to Java’s native serialization interface described above. Additionally, serialization was chosen over externalization for the same reason (Baeldung, 2021).

Avenues for Development

Recommendation System

While not fully utilized by the system, the student objects store a count of all valid genres that they have borrowed along with the number of borrows & returns. These data would be better suitably stored in a student profile attribute so that they can be combined with the profiles of other students. Such a setup will allow for recommending books based on their most frequently borrowed, or late returned genres based on a student's profile or the aggregated data across all students. Where the rationale for allowing both, would be to manage the event that a single student has a limited rental history, but a collection of similar students collectively have a large rental history that can better inform recommendations, or a new student can view the system by most popular genres.

Book Genre Enum: Inventory Maintenance & Searching

The Genre enum was developed to allow the system to later expose searching books by genre, but also allow an admin to apply a valid genre to a book missing this data, or request an extension to the enum with new genre types. The use of the genre enum was the solution to the problem that there was a collection of different null values in the initial mock data, but all other genres were consistent. It allowed for ranking these objects numerically to simplify potential sorting, where null values are 0 and the max value is currently 17. It also allowed for books to be constructed with a set of valid genres that are consistent throughout the system. However, the use of this enum is currently utilized by the LDBMS because the solution breaks down from a user experience perspective if they wish to add a new genre. Additionally, searching by genre requires the creation of a new index that uses a bookID as a foreign key, and separate index entries for each genre of the book with the primary key being the foreign key.

Summary

The modular design that was planned from modeling the LDBMS from a database & user interaction point of view greatly simplified the development process and helped establish an outline for the MVC-like scheme. The design allowed for separate components for the current prototype to be explored separately but with sufficient time to identify design limitations that can shape the next directions. That collectively helped steer the design trajectory to cover the main requirements for the client’s brief. Where the most fruitful next steps include refactoring the dataset chain to a hash map so that the respective modules can be more easily maintained, developing an enum to support & simplify the index entry comparisons and index searching algorithms, and adding a tree data structure to the index interface so that new data can be added in a sorted manner without having to rebuild the indexes or restart the system, and finalize the client’s preference on constants & specific datapoint attributes that they need before exploring the avenues for development discussed in “*Avenues for Development*”.

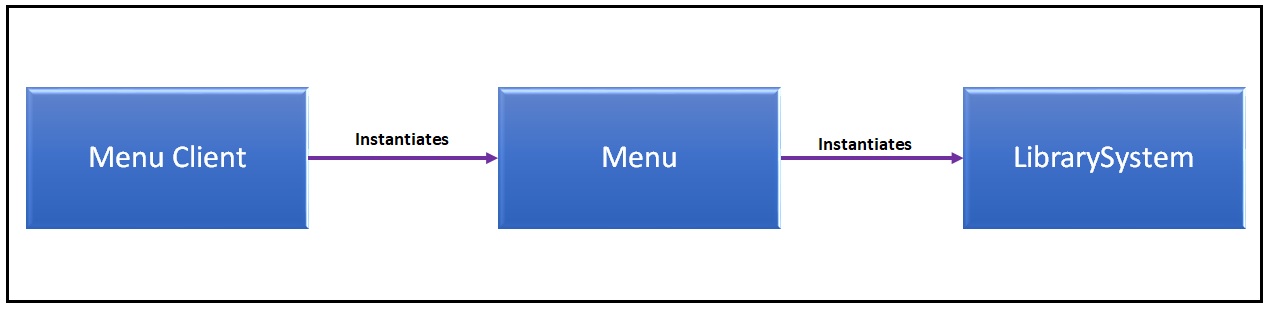
References

Baeldung. (2021). *Different Serialization Approaches for Java*. https://www.baeldung.com/java-serialization-approaches

Sciore, E. (2019). *Java Program Design: Principles, Polymorphism, and Patterns* (A. Jecan (ed.)). Apress. https://doi.org/https://doi.org/10.1007/978-1-4842-4143-1

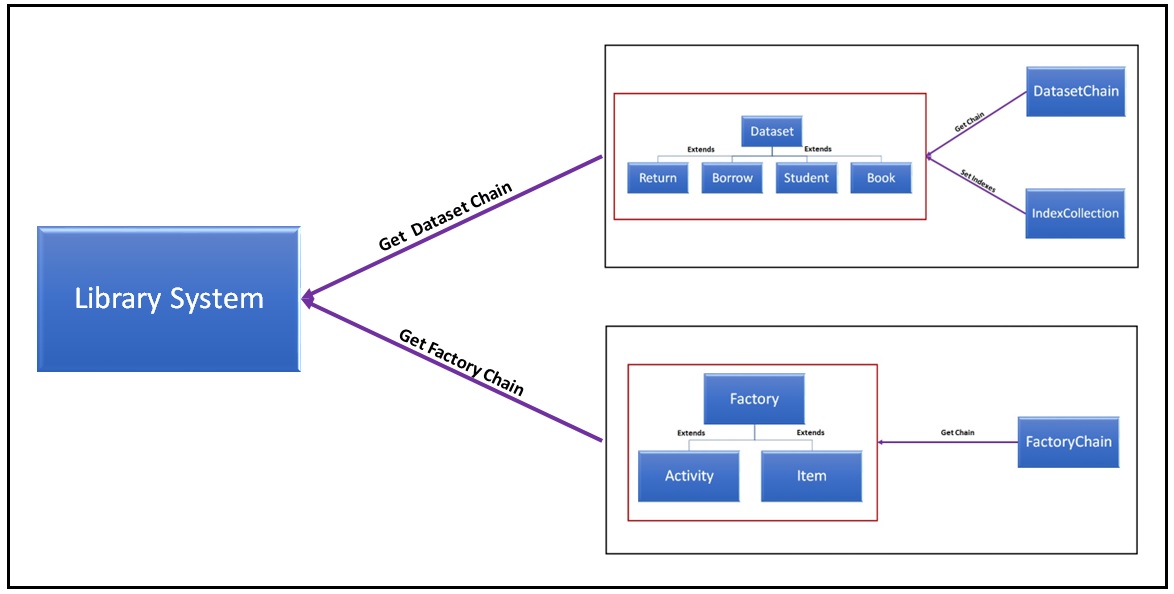
Figure Appendices

Figure 1: System Design



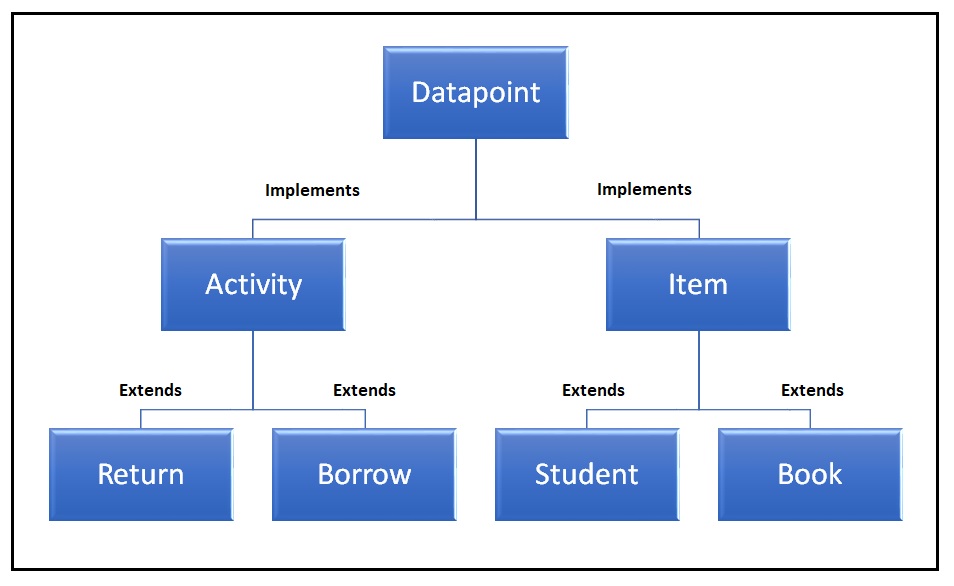
*Displays how the view leverages the library system controller for CRUD operations against the LDBMS. The library system uses is composed of collection-based supporting modules for mediating interactions between the view and model, as shown in the collective figure appendices.*

Figure 2: Library System



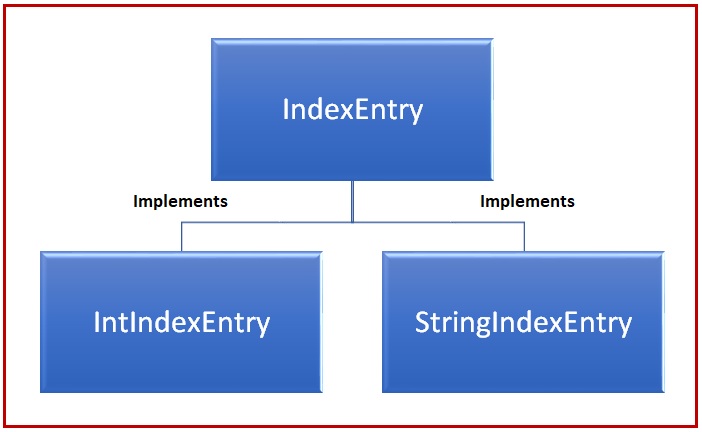
*Displays how the library system controller uses the supporting collection/chain-based objects to manipulate models of the system on behalf of the client/view.*

Figure 3: Datapoint Interface Hierarchy



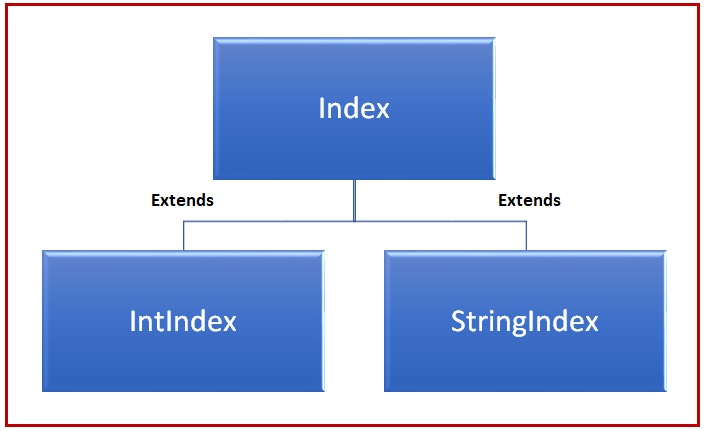
*Displays the hierarchy of the LDBMS core model classes. The abstract Item class represents tangible objects of the LDBMS (i.e Users/Students and Inventory/Books). The abstract Activity class represents interactions between these objects as Borrows and Returns.*

Figure 4: Index Entry Interface Hierarchy



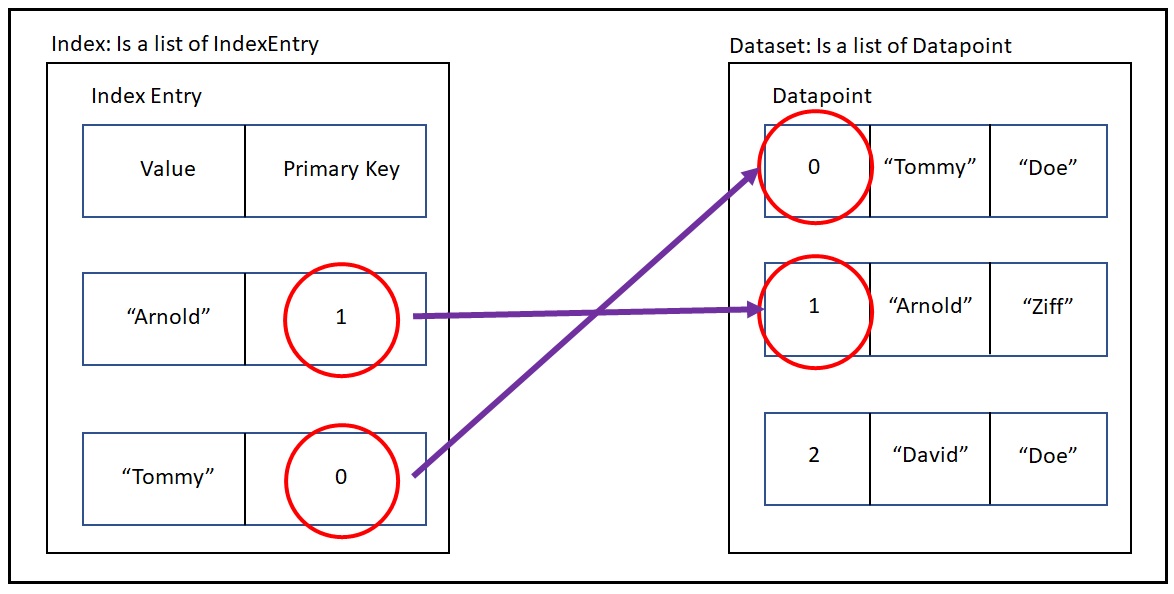
*Displays the hierarchy of the index entry interface. Concrete objects have a property representing a value position in an array, and the value at that position. Custom comparison methods etc are overridden on the concrete level.*

Figure 5: Abstract Index Hierarchy



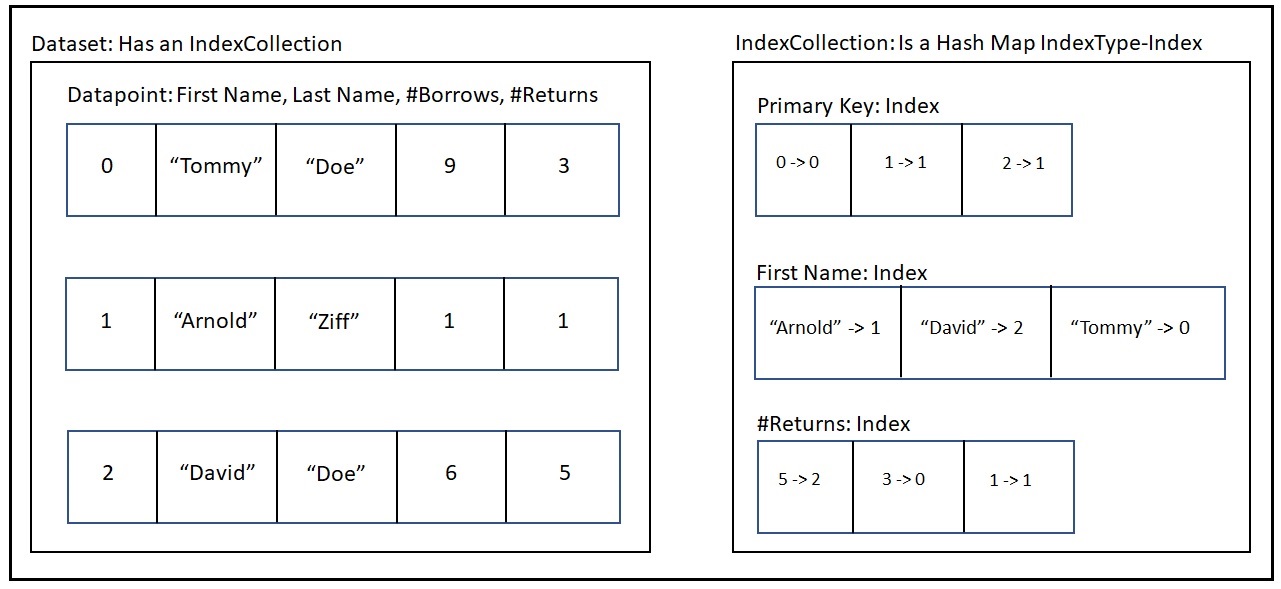
*Displays the hierarchy of the index module. The index is an abstract class allowing for generic searching & sorting methods, while also allowing for any additional operations to be overridden by the concrete classes that extend.*

Figure 6: Principle of the Index Interface



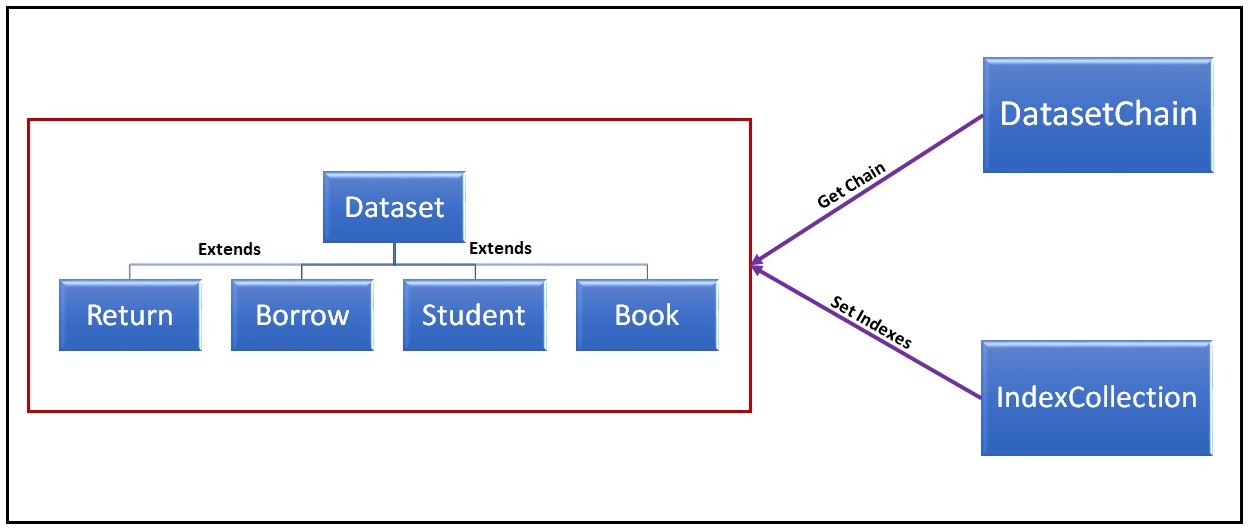
*Graphical representation of the principle of how the index interface is used for an array. Where an index is a list of index entries (string/int) that contain a pointer to a position in an array and the value of the array at that position. The index entries manage comparisons and the index manages searching & sorting.*

Figure 7: Principle of the Index Collection



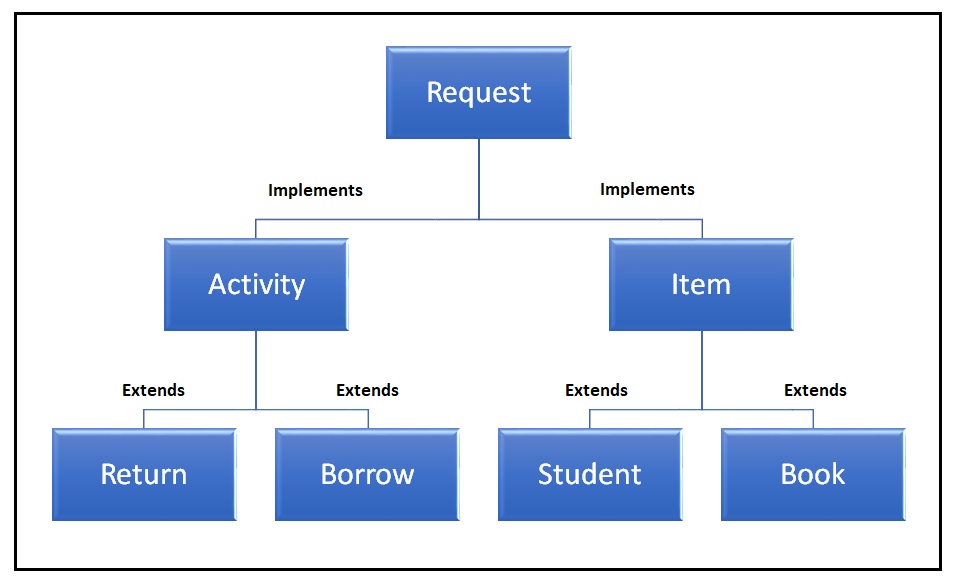
*Graphical representation of the principal of the index collection. It is a hash map of valid dataset indexes managed by an enum, whose values are a string/integer index corresponding to the attribute of the datapoint. The dataset module uses the index collection module for querying appropriate indexes.*

Figure 8: Abstract Dataset Chain Hierarchy



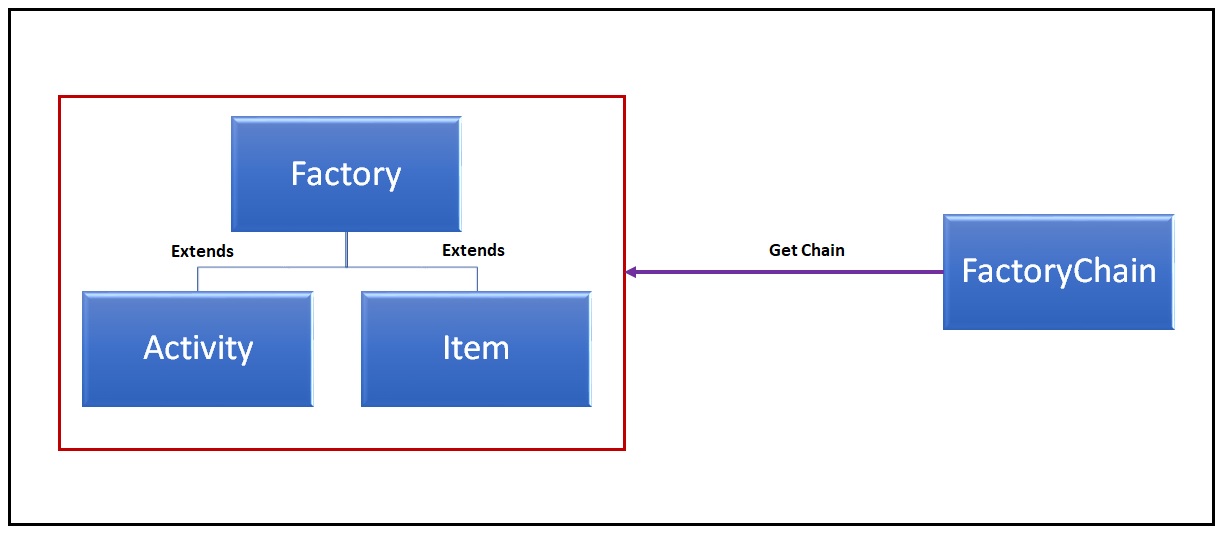
*The abstract dataset chain uses dataset levels to accept/delegate CRUD operations against it. It is composed of a list of data points and an index collection querying related indexes as per table 3.*

Figure 9: Request Interface Hierarchy



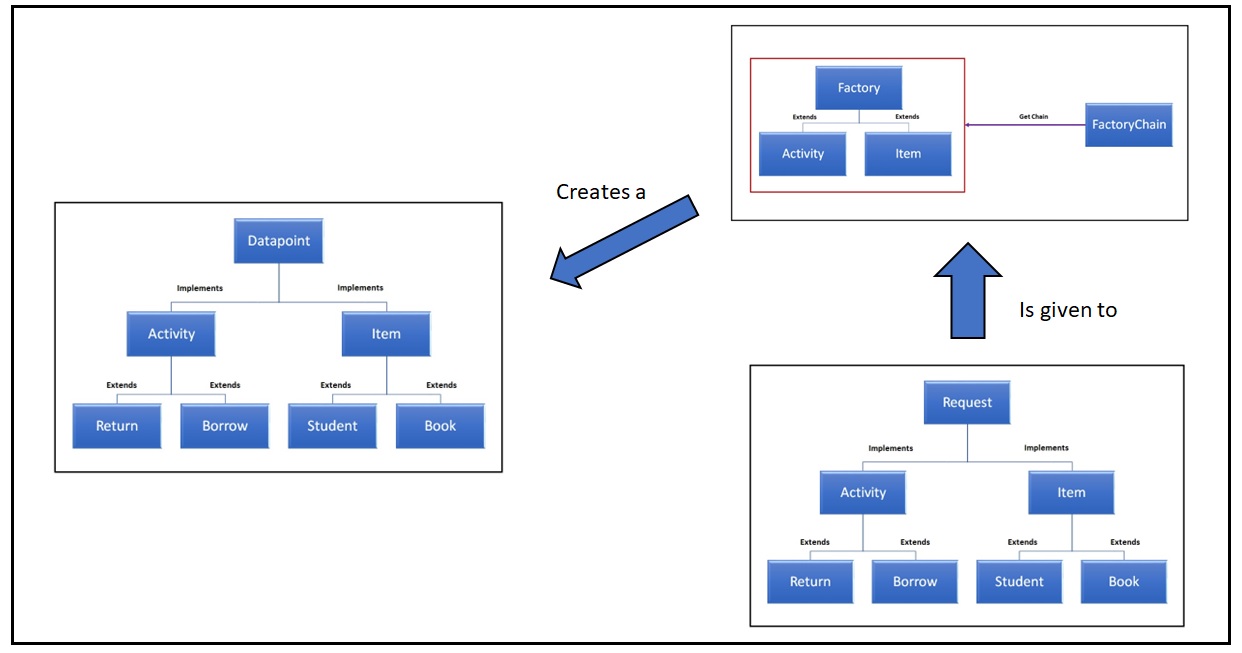
*The request interface is used to coordinate with the factory chain for concrete datapoint construction. Concrete request objects are hash maps that store the values for datapoint attributes. Activity requests are responsible for validating whether related data points can be constructed (book already borrowed, student borrow limit reach) and handing the dates (configuration, evaluating returns as late/on time).*

Figure 10: Abstract Factory Chain



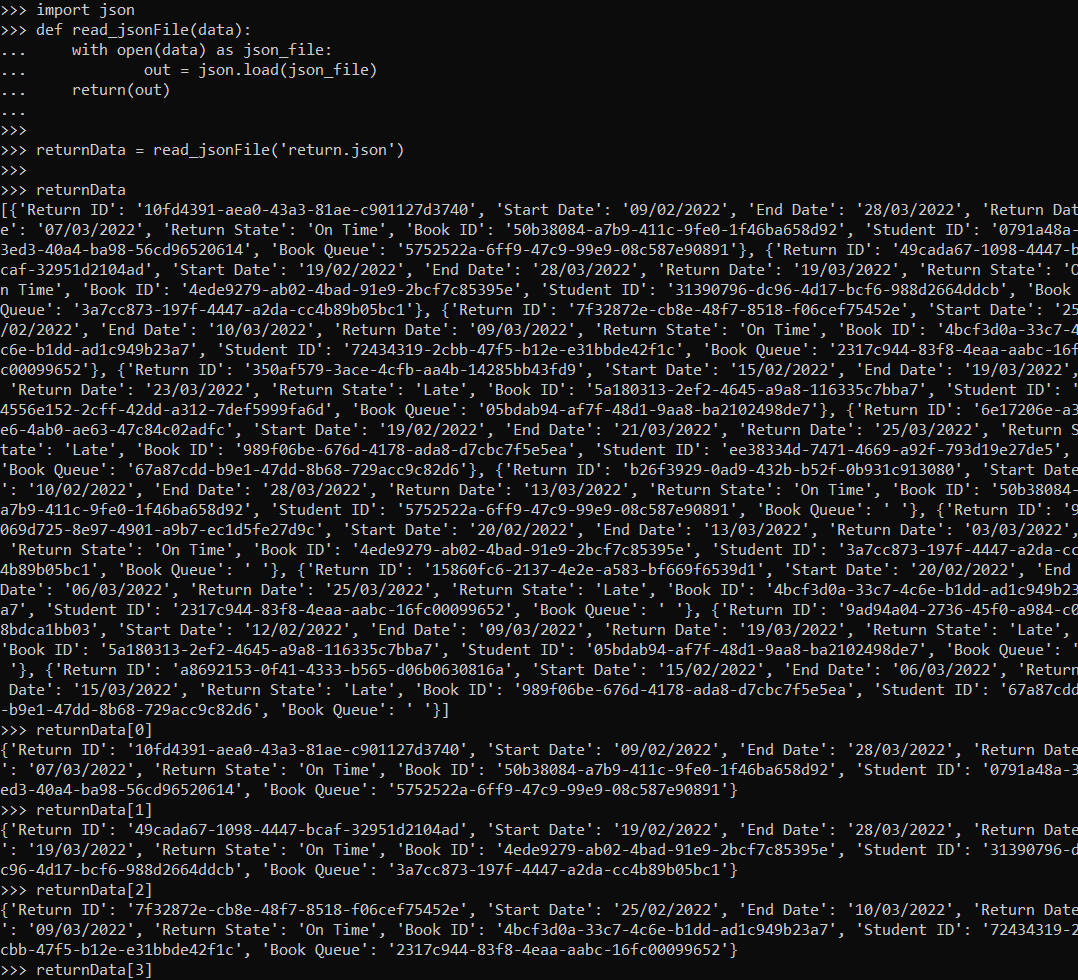
*Datapoint factory is an abstract class where the Item and Activity factories are responsible for constructing the appropriate child via the datapoint edge enum. The circular chain of command allows the factories to accept the construction of data points on its level, or delegate the request. The factory class is responsible for configuring the chain and passing it to the class that uses it.*

Figure 11: Datapoint Construction



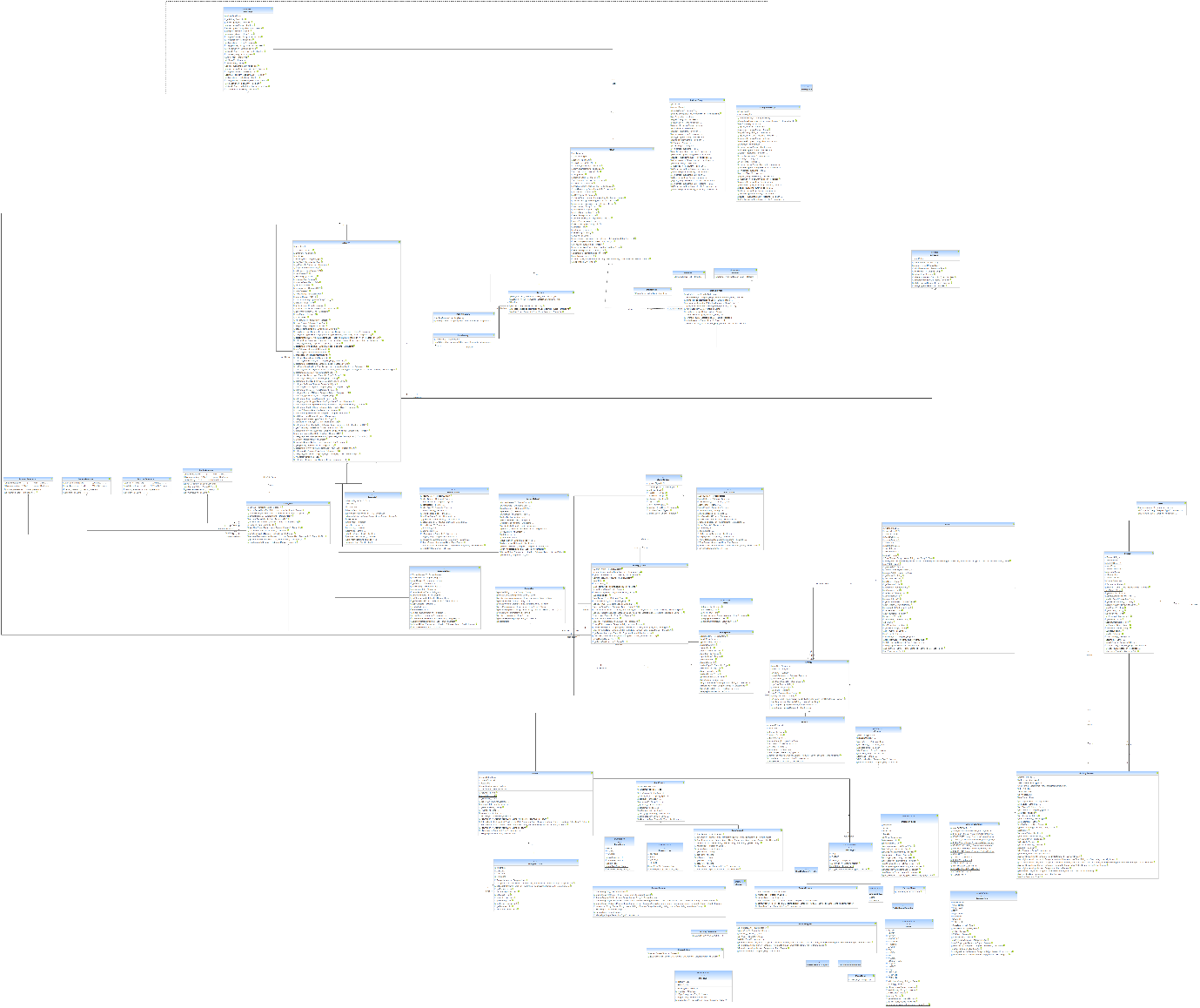
*Demonstration of datapoint construction from the hierarchal perspective. The circular abstract factory chain uses a request object to hold datapoint specific data and a datapoint edge enum value to construct the datapoint. The use of the request interface allowed the datapoint edge enum to construct concrete classes, while the use of constants allows the Item/Activity factories to accept requests or delegate them to the next factory.*

Figure 12: Example JSON Export



*Demonstration that the custom implementation of JSON exports is usable by other software.* ***NB:*** *Implementation has not been updated/verified since the development of custom array list queues on books, array set on students, and use of integers for datapoint IDs.*

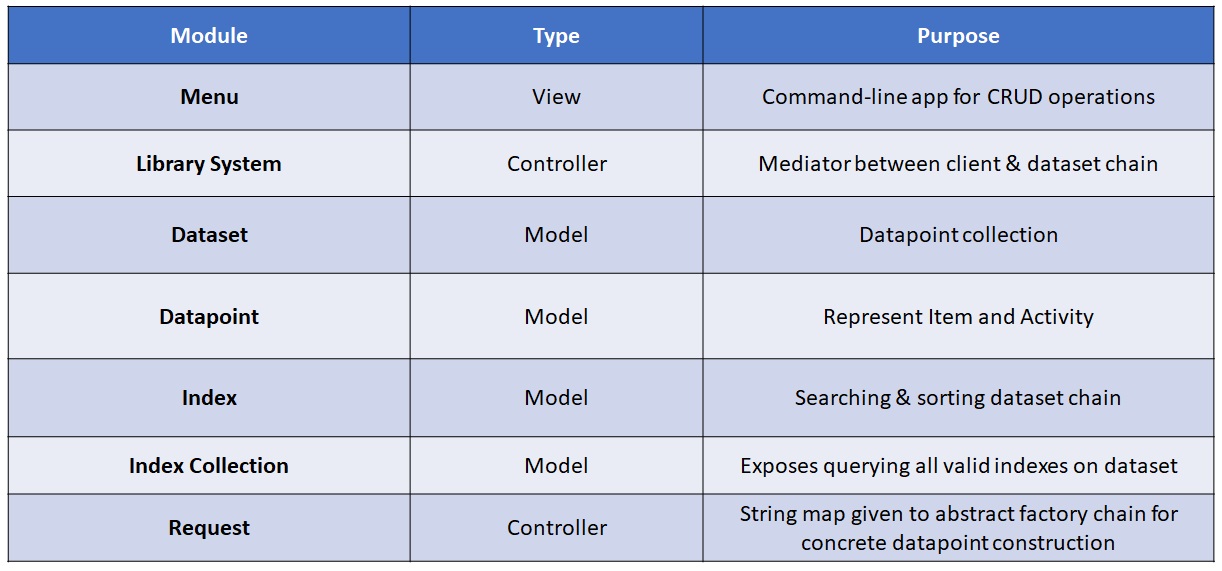
Figure 13: Library Database Management System Class Diagram



*Owing to the lack of readability in this document due to its size the class diagram and related data from “Yatta UML Lab” are located in the “resources” directory of the “built” project that was submitted.* ***NB:*** *The tool was unable to include Enums, relate menu client to library menu, and relate constants classes to the classes that use them. Which was common among other tested tools: Object Aid & the BPMN2 Eclipse plugins.*

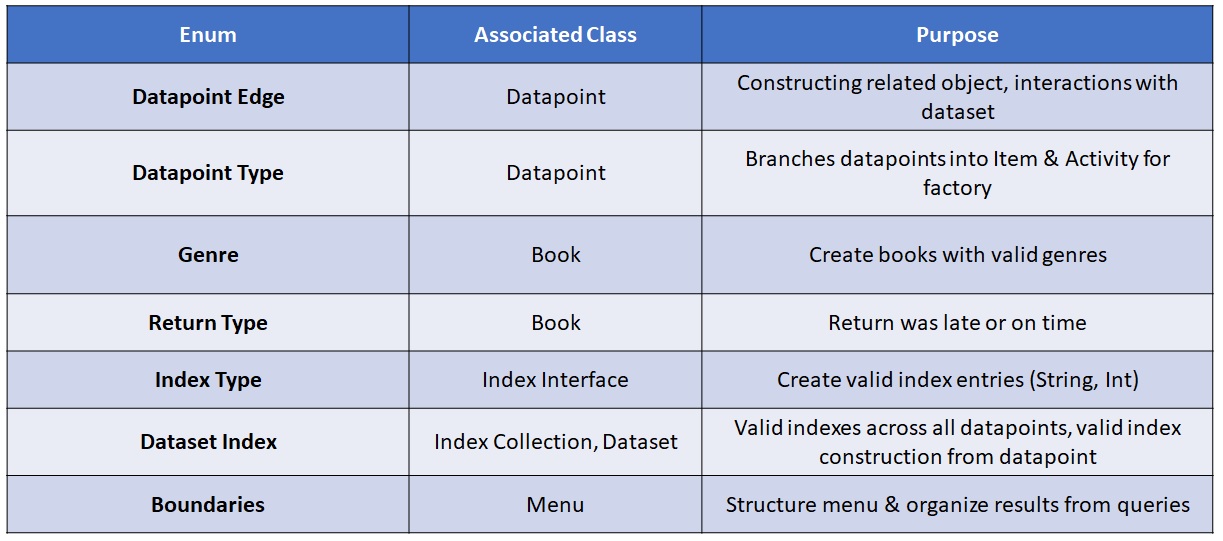
Table Appendices

Table 1: Core Modules



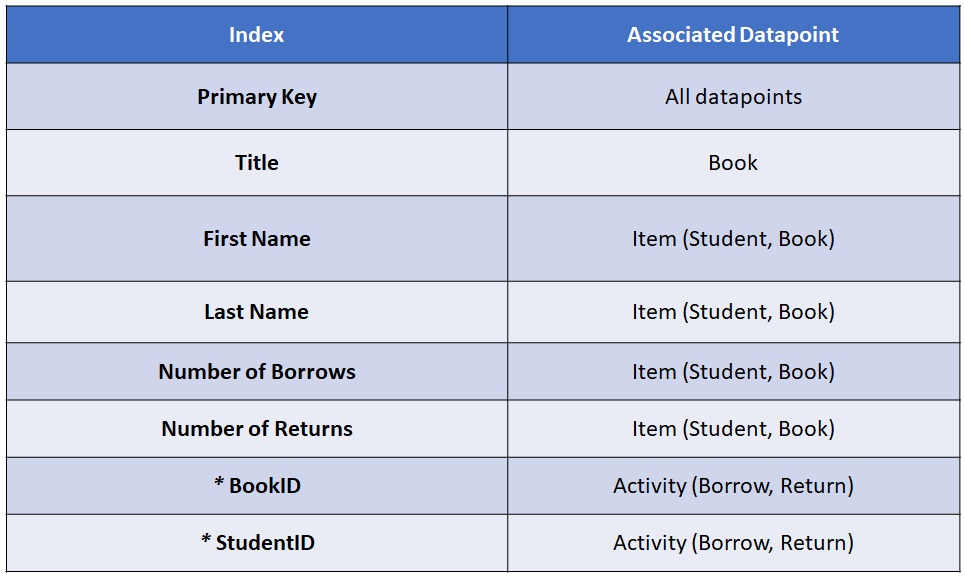
*Table to summarize where core modules place in the LDBMS MVC design. Where collections (ie dataset) were viewed as a supporting class for the systems models. Requests are viewed on the controller level because they support the controller in mediating communications between client/view and model.*

Table 2: Enums Supporting System



*Table to display all core enums used in the LDBMS, their relevance, and modules that utilize them.*

Table 3: Indexed Datapoint Attributes



*Displays the enum values from the “Dataset\_Index” class and their associated data point. \*Represent foreign keys that used to develop were implemented in the menu for non-distinct joins to display active & historical activities on the Item level (ie Book or Student).*