**Gathering Requirements**

Gathering the functional and non-functional requirements of a Library Management System is the starting point in designing the system.

**Functional Requirements**

**What are functional requirements?**

Functional requirements are like a set of instructions for a system. They describe what the system should do and how it should do it. They cover what the system should accept as input, what it should produce as output, and how it should interact with users and other methods. These requirements are essential because they help ensure that the result is what everyone wants and needs.

**Functional Requirements for a Library Management System**

The following are the functional requirements for a Library Management System:

1. Ability to add and remove books from the library
2. Ability to search for books in the library by title, author, or ISBN
3. Ability to check out and return books
4. Ability to display a list of all books in the library
5. Ability to store and retrieve information about library patrons, including their name and ID number
6. Ability to track which books are currently checked out and when they are due to be returned
7. Ability to generate reports on library usage and checkouts

**Non-Functional Requirements**

**What are non-functional requirements?**

Non-functional requirements describe how a system should work, not just what it should do. They include how fast it should run, how secure it should be, and how easy it should be to use. They help make sure the system works well and meets people's expectations.

**Non-Functional Requirements for a Library Management System**

The following are the non-functional requirements for a Library Management System:

1. User-friendly interface for easy navigation and use
2. High performance and scalability to handle large amounts of data
3. Data security and protection to ensure the privacy and confidentiality of library patrons and their information
4. Compatibility with various operating systems and devices
5. Ability to handle multiple users and concurrent access to the system
6. Compliance with relevant laws and regulations regarding library management and data privacy
7. Regular updates and maintenance to ensure the system remains functional and secure over time.

**Entities and their Relationships**

**Entities**

**What are entities?**

In system design, entities are objects or concepts with distinct characteristics. They represent real-world things and are used to organize data within the system. Entities provide a structure for the data and relationships and define the data fields and connections within the system.

**Entities for a Library Management System**

The following are the entities for a Library Management System:

1. **Book:**Each book in the library is represented by a Book entity.
2. **Library:**The collection of books in the library is represented by the Library entity.
3. **Patron:** Each library patron is represented by a Patron entity.
4. **Checkout:** Each checkout transaction is represented by a Checkout entity.

**Relationships**

**What are relationships?**

In system design, a relationship between two entities shows how the data is organized and related. It is defined by the number of entities involved, the type of relationship, the direction of the relationship, and any rules that govern it.

**How will we define relationships?**

In system design, relationships between entities are defined to show how the data is connected. This is done by:

* Specifying how many entities are involved (one-to-one, one-to-many, many-to-many).
* Describing the type of relationship (aggregation, composition, inheritance).
* Determining the direction of the relationship (unidirectional or bidirectional).
* Defining rules for the relationship (mandatory/optional, unique/non-unique).

**Relationships for a Library Management System**

The relationships for a Library Management System are as follows:

1. A Book can be added or removed from a Library.
2. A Book can be checked out or returned by a Patron.
3. A Patron can check out multiple Books, and multiple Patrons can check out each Book.
4. A Checkout is associated with a specific Book and Patron and records the date when the book is due to be returned.
5. A Library has a collection of Books.
6. A Patron can check out multiple books and return multiple books.
7. A Book can be associated with multiple checkouts.

**Functional Decomposition**

**What is functional decomposition?**

Functional Decomposition is breaking down a complex function or system into smaller and simpler sub-functions or components.

**Functional Decomposition for a Library Management System**

Here is a functional decomposition of the library management system based on the functional requirements mentioned:

1. Add/Remove Books
   1. AddBook(Book book)
   2. RemoveBook(string ISBN)
2. Search Books
   1. SearchBookByTitle(string title)
   2. SearchBookByAuthor(string author)
   3. SearchBookByISBN(string ISBN)
3. Checkout/Return Books
   1. CheckOutBook(Book book, Patron patron)
   2. ReturnBook(Book book)
4. Display Books
   1. DisplayBooks()
5. Patron Management
   1. AddPatron(Patron patron)
   2. RemovePatron(string ID)
   3. EditPatron(Patron patron)
6. Checkout Management
   1. DisplayCheckouts()
   2. GetDueDate(Book book)
7. Report generation
   1. GenerateCheckoutReport()
   2. GeneratePatronReport()
   3. GenerateBookReport()

Each function has a specific task and takes inappropriate inputs, processes the data, and returns the output. This way, the system can be easily understood and maintained. Additionally, this functional decomposition can help identify any functionality or requirements in the system.

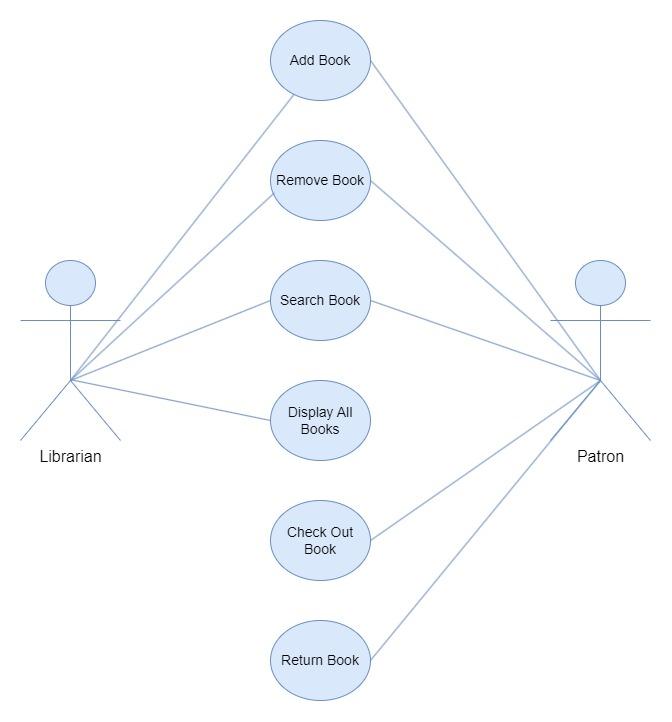
**Class Diagram**

A class diagram is like a map that shows the different parts and relationships of a system. It helps developers plan and design the structure of a system.

For a comprehensive understanding of class diagrams, including what they are and how they are created, please refer to [this](https://www.codingninjas.com/studio/library/what-is-class-diagram) article.

**Use Case Diagram**

A use case diagram is like a flowchart that shows how different people (called actors) can use a system to do different things. It helps developers understand what the system needs to do and how it will be used.



For a full understanding of use-case diagrams, including their definition and the process for creating them, please refer to [this](https://www.codingninjas.com/studio/library/what-is-use-case-diagram) article.

**Low-Level Design of Library Management System Classes and Methods Involved**

Low-level design for a library management system in C++ would involve creating classes and functions to handle the various operations and data required for the system. Here is an example of how the system could be designed:

1. **Book Class:** This class would store information about a single book, such as its title, author, ISBN number, and availability status. It would also include functions to check out and return a book.
2. **Library Class:**This class would store a collection of books and handle operations such as adding and removing books, searching for books, and displaying the list of books.
3. **Patron Class**: This class would store information about library patrons, such as their names and ID numbers. It would also include functions to check out and return books.
4. Checkout Class: This class would handle the checking out and returning of books. It would store information about which patron has checked out which book and when it is due to be returned.
5. main() function: This function would be responsible for creating objects of the Library, Book, Patron, and Checkout classes and calling the appropriate functions to carry out the library management tasks.

**Here is an example of how the Book class could be implemented in C++:**

class Book {

private:

string title;

string author;

string ISBN;

bool availability; // true if available, false if checked out

public:

// constructor to initialize book's information

Book(string t, string a, string i) {

title = t;

author = a;

ISBN = i;

availability = true;

}

// function to check out a book

void checkOut() {

if (availability) {

availability = false;

cout << "Book has been checked out." << endl;

} else {

cout << "Book is not available." << endl;

}

}

// function to return a book

void returnBook() {

if (!availability) {

availability = true;

cout << "Book has been returned." << endl;

} else {

cout << "Book has not been checked out." << endl;

}

}

// function to get book's title

string getTitle() {

return title;

}

// function to get book's author

string getAuthor() {

return author;

}

// function to get book's ISBN

string getISBN() {

return ISBN;

}

// function to check if book is available

bool isAvailable() {

return availability;

}

};

You can also try this code with **Online C++ Compiler**

[**Run Code**](https://www.codingninjas.com/studio/online-compiler/online-cpp-compiler)

This class stores the book's title, author, ISBN number, and availability status as private member variables. It has public member functions for checking out and returning a book, as well as functions to access the book's title, author, ISBN, and availability status.

**Here is an example of how the Library class could be implemented in C++:**

class Library {

private:

vector<Book> books;

public:

// function to add a book to the library

void addBook(Book b) {

books.push\_back(b);

cout << "Book has been added to the library." << endl;

}

// function to remove a book from the library

void removeBook(string ISBN) {

for (int i = 0; i < books.size(); i++) {

if (books[i].getISBN() == ISBN) {

books.erase(books.begin() + i);

cout << "Book has been removed from the library." << endl;

return;

}

}

cout << "Book with ISBN " << ISBN << " not found in library." << endl;

}

// function to search for a book in the library

Book\* searchBook(string ISBN) {

for (int i = 0; i < books.size(); i++) {

if (books[i].getISBN() == ISBN) {

return &books[i];

}

}

cout << "Book with ISBN " << ISBN << " not found in library." << endl;

return nullptr;

}

// function to display all books in the library

void displayBooks() {

cout << "Books in the library: " << endl;

for (int i = 0; i < books.size(); i++) {

cout << "Title: " << books[i].getTitle() << endl;

cout << "Author: " << books[i].getAuthor() << endl;

cout << "ISBN: " << books[i].getISBN() << endl;

cout << "Availability: " << (books[i].isAvailable() ? "Available" : "Checked out") << endl;

cout << endl;

}

}

};

**Here is an example of how the Patron class could be implemented in C++:**

#include <vector>

class Patron {

private:

string name;

string ID;

vector<Book\*> checkedOutBooks;

public:

// constructor to initialize patron's information

Patron(string n, string i) {

name = n;

ID = i;

}

// function to check out a book

void checkOutBook(Book \*book) {

if (book->isAvailable()) {

book->checkOut();

checkedOutBooks.push\_back(book);

cout << "Book has been checked out to " << name << "." << endl;

} else {

cout << "Book is not available." << endl;

}

}

// function to return a book

void returnBook(Book \*book) {

for (int i = 0; i < checkedOutBooks.size(); i++) {

if (checkedOutBooks[i]->getISBN() == book->getISBN()) {

book->returnBook();

checkedOutBooks.erase(checkedOutBooks.begin() + i);

cout << "Book has been returned by " << name << "." << endl;

return;

}

}

cout << name << " did not check out this book." << endl;

}

// function to get patron's name

string getName() {

return name;

}

// function to get patron's ID

string getID() {

return ID;

}

};

You can also try this code with **Online C++ Compiler**

[**Run Code**](https://www.codingninjas.com/studio/online-compiler/online-cpp-compiler)

This class stores the patron's name, ID number, and a vector of pointers to the books that the patron has checked out as private member variables. It has public member functions for checking out and returning books, as well as functions to access the patron's name and ID. The checkOutBook function accepts a pointer to a book object, checks if it's available, and checks it out if it is. The returnBook function accepts a pointer to a book object, searches for it in the vector of checkedOutBooks, and returns it if found.

**Here is an example of how the Checkout class could be implemented in C++:**

class Checkout {

private:

map<Book\*, pair<Patron\*, time\_t>> checkouts;

const int DAY\_SECONDS = 60 \* 60 \* 24;

const int MAX\_FINE = 10;

public:

// function to check out a book

void checkOut(Book \*book, Patron \*patron) {

if (book->isAvailable()) {

patron->checkOutBook(book);

time\_t dueDate = time(0) + 60 \* 60 \* 24 \* 7; // 7 days from current time

checkouts[book] = make\_pair(patron, dueDate);

} else {

cout << "Book is not available." << endl;

}

}

// function to return a book

void returnBook(Book \*book) {

if (checkouts.count(book) == 1) {

Patron \*patron = checkouts[book].first;

patron->returnBook(book);

checkouts.erase(book);

} else {

cout << "Book has not been checked out." << endl;

}

}

// function to display all current checkouts

void displayCheckouts() {

cout << "Current checkouts: " << endl;

for (auto const& x : checkouts) {

cout << "Book: " << x.first->getTitle() << endl;

cout << "Author: " << x.first->getAuthor() << endl;

cout << "Patron: " << x.second.first->getName() << endl;

cout << "Due Date: " << ctime(&x.second.second);

cout << endl;

}

}

time\_t getDueDate(Book \*book) {

if (checkouts.count(book) == 1) {

return checkouts[book].second;

}

else {

cout << "Book has not been checked out." << endl;

return -1;

}

}

// function to calculate the fine for a book that is overdue

double calculateFine(Book\* book) {

time\_t dueDate = this->getDueDate(book);

time\_t currentDate = time(0);

double secondsLate = difftime(currentDate, dueDate);

if (secondsLate < 0) {

// book was returned early, no fine

return 0.0;

} else {

// calculate fine based on how many days late the book is

int daysLate = secondsLate / DAY\_SECONDS;

double fine = daysLate \* 0.50;

// cap the fine at MAX\_FINE

return min(fine, (double)MAX\_FINE);

}

}

};

You can also try this code with **Online C++ Compiler**

[**Run Code**](https://www.codingninjas.com/studio/online-compiler/online-cpp-compiler)

This class stores a map of Book pointer as key and a pair of Patron pointer and time\_t as value, representing the checkouts. It has public member functions for checking out and returning books, as well as a function to display all current checkouts. The checkOut function accepts a pointer to a book object and a pointer to a patron object, checks if the book is available, checks it out if it is, and stores the current time as the due date.

The returnBook function accepts a pointer to a book object and searches for it in the map of checkouts, returning it if found. The displayCheckouts function displays all the checkouts by iterating over the map and displaying the book's title, author, patron's name, and due date.

**Here is an example of how the main() function could be implemented in C++ to use the previously defined classes:**

int main() {

// create objects of Library, Book, Patron, and Checkout classes

Library library;

Book book1("The Great Gatsby", "F. Scott Fitzgerald", "1234567890");

Book book2("To Kill a Mockingbird", "Harper Lee", "0987654321");

Patron patron1("John Smith", "123");

Patron patron2("Jane Doe", "456");

Checkout checkout;

// add books to the library

library.addBook(book1);

library.addBook(book2);

// display all books in the library

library.displayBooks();

// check out a book

checkout.checkOut(&book1, &patron1);

// get the due date of the checked out book

time\_t dueDate = checkout.getDueDate(&book1);

if (dueDate != -1) {

cout << "Due Date: " << ctime(&dueDate) << endl;

}

// calculate and display the fine for the overdue book

double fine = checkout.calculateFine(&book1);

if (fine > 0) {

cout << "Fine for overdue book: Rs" << fine << endl;

}

// display all current checkouts

checkout.displayCheckouts();

// return a book

checkout.returnBook(&book1);

// display all current checkouts

checkout.displayCheckouts();

return 0;

}

You can also try this code with **Online C++ Compiler**

[**Run Code**](https://www.codingninjas.com/studio/online-compiler/online-cpp-compiler)

**Data Structures and Algorithms Used**

Here are some examples of data structures and algorithms that could be used in the library management system:

1. **Books:** A vector of Book objects could be used to store all the books in the library. This data structure allows for easy addition and removal of books and efficient iteration through the collection of books. A hashmap can be used to store books with ISBN as a key, it allows for faster searching of books by ISBN
2. **Patrons:** A vector of Patron objects could be used to store all the patrons in the library. This data structure allows for easy addition and removal of patrons, as well as an efficient iteration through the collection of patrons. A hashmap can be used to store patrons with ID as a key, it allows for faster searching of patrons by ID
3. **Checkouts:** A map can store the checkouts, with the Book object as the key and a pair of Patron objects, and time\_t as the value. This data structure allows for easy retrieval of the patron and due date associated with a specific book.
4. **Searching:** Linear search algorithm could be used to search for books by title and author. A binary search algorithm could be used if the books are stored in sorted order.
5. **Sorting:** Insertion sort or bubble sort algorithms could be used to sort the books in alphabetical order by title or author.

It's important to note that the data structures and algorithms mentioned are examples and other options that better suit the system's needs could be used. The choice of data structures and algorithms depends on the data size, operation complexity, and desired performance of the system.