short$_n ote(extra)$

Rudra Nepal

August 2024

1 Responsibility-Driven Design (RDD)

Responsibility-Driven Design (RDD) is an approach to design that emphasizes behavioral modeling using objects, responsibilities, and collaborations. In this model, objects play specific roles and occupy well-defined positions in the application architecture. Each object is responsible for a specific portion of the work, collaborating with other objects to achieve the larger goals of the application. This approach builds a collaborative model of the application through a "community of objects," with clearly defined responsibilities.

1.1 Library Management System Example

1.2 1. Identify Candidate Objects

- **BOOK**
- **LIBRARY**
- **PATRON**

1.3 2. Assign Responsibilities

- **Book Class Responsibilities:**
  - Keep track of its title, author, and availability status.
  - Provide methods to access and update this information.
- **Library Class Responsibilities:**
  - Maintain a collection of books.
  - Implement methods for adding, removing, and searching books.
  - Handle borrowing and returning of books.
- **Patron Class Responsibilities:**
  - Request to borrow a book from the library.
  - Return a borrowed book to the library.
  - Search for a book in the library.

1.4 3. Define Collaborations

- **Patron** sends a `requestBorrow` message to the **Library**, which checks the book's availability and updates it accordingly.
- **Patron** sends a `returnBook` message to the **Library**, which updates the availability status of the returned book.
- **Patron** sends a `searchBook` message to the **Library**, which searches for the requested book and returns the result.

2 Responsibility Implies Noninterference

Conventional programming often involves performing actions on various elements of the system, such as modifying records or updating arrays. This can lead to sections of code being closely tied through control and data connections. Dependencies may arise from global variables, pointers, or inappropriate reliance on the implementation details of other code sections.

Responsibility-Driven Design (RDD) aims to minimize these connections, making them as unobtrusive as possible. By assigning specific responsibilities to objects (or components) within a system, we expect certain behaviors when rules are followed. Crucially, responsibility implies a degree of independence or noninterference. For instance, when we assign a child the responsibility of cleaning her room, we typically do not oversee every action she takes. Instead, we expect that by issuing a directive properly, the desired outcome will be achieved without constant supervision.

3 Programming in Small and Large

The distinction between developing individual projects and larger software systems is often framed as programming in the small versus programming in the large.

3.1 Programming in the Small

Programming in the small is characterized by the following attributes:

- Code is developed by a single programmer or a very small team. A single individual can understand all aspects of the project, from start to finish.
- The major challenge is the design and development of algorithms to address the problem at hand.

3.2 Programming in the Large

Programming in the large, by contrast, involves features such as:

- The software system is developed by a large team, which may include graphic artists, design experts, and programmers. Individuals involved in specification, design, coding, and integration may differ, and no single person is responsible for the entire project or understands all its aspects.
- The primary challenge is managing details and facilitating communication between diverse components of the project.

4 Standard Template Library (STL)

The Standard Template Library (STL) is a powerful set of C++ template classes that provide general-purpose data structures and algorithms. It is a fundamental part of the C++ Standard Library and is designed to improve productivity and code efficiency by offering reusable components that are both efficient and easy to use.

4.1 Components of STL

The STL consists of four main components:

- **Containers**: These are data structures that store objects and data. STL provides several types of containers including:
  - `vector`: A dynamic array that can grow and shrink in size.
  - `list`: A doubly linked list allowing fast insertions and deletions.
  - `deque`: A double-ended queue with efficient insertions and deletions at both ends.
  - `set` and `map`: Associative containers that store elements in a sorted order and allow fast retrieval based on keys.
  - `unordered_set` and `unordered_map`: Hash-based containers that provide average constant-time complexity for insertions and lookups.
- **Algorithms**: STL provides a range of algorithms that operate on containers, such as:
  - `sort()`: Sorts the elements of a container.
  - `find()`: Searches for an element in a container.
  - `accumulate()`: Computes the sum of elements.
  - `count()`: Counts occurrences of a specific value.
- **Iterators**: These are objects that provide a way to access the elements of containers in a sequential manner. They act as pointers and support operations such as increment and dereference. Types of iterators include:
  - `begin()`: Returns an iterator to the first element.

  - `end()`: Returns an iterator to one past the last element.

- **Function Objects (Functors)**: These are objects that can be called as if they were functions. They are used to define custom operations for algorithms. For example, the `std::greater` functor is used to specify a comparison operation for sorting in descending order.

4.2 Advantages of STL

- **Reusability**: STL components are generic and reusable across different programs.
- **Efficiency**: STL provides efficient implementations of common data structures and algorithms.
- **Consistency**: STL offers a consistent interface for various containers and algorithms.

5 Composition Relationship Diagram

In C++, a composition relationship represents a strong "part-of" relationship between two classes. In this relationship, the lifetime of the part (contained object) is tied to the lifetime of the whole (container object). The contained object is created and destroyed with the container object and cannot exist independently of it.

5.1 Composition Relationship

- **Container Class**: The class that contains other classes (the whole).
- **Contained Class**: The class that is contained within another class (the part).
- **Lifetime**: The lifetime of the contained class is tied to the lifetime of the container class.

5.2 Explanation of the Diagram

- **Container Class**: Represents the class that contains other objects. In this example, it is labeled `Container`.
- **Part Class**: Represents the class that is contained within the `Container`. In this example, it is labeled `Part`.
- **Composition Relationship**: The filled diamond on the line connecting `Container` and `Part` signifies that `Container` has a composition relationship with `Part`. This means `Part` objects are created and destroyed with `Container` objects.

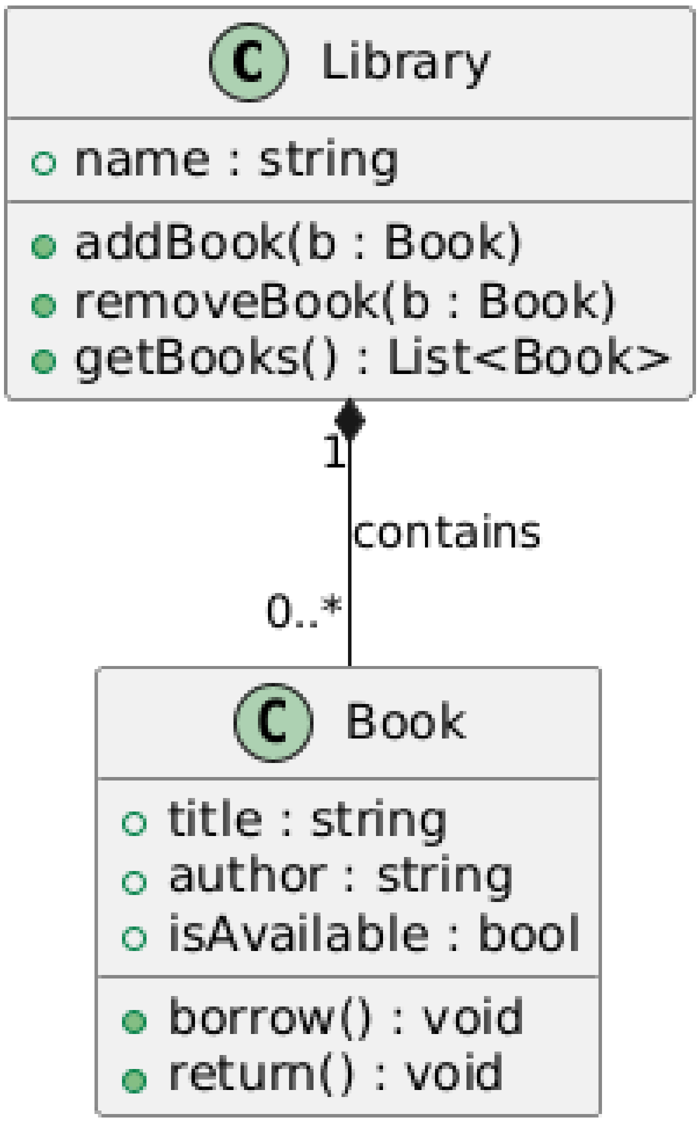

Figure 1: Composition Relationship Diagram