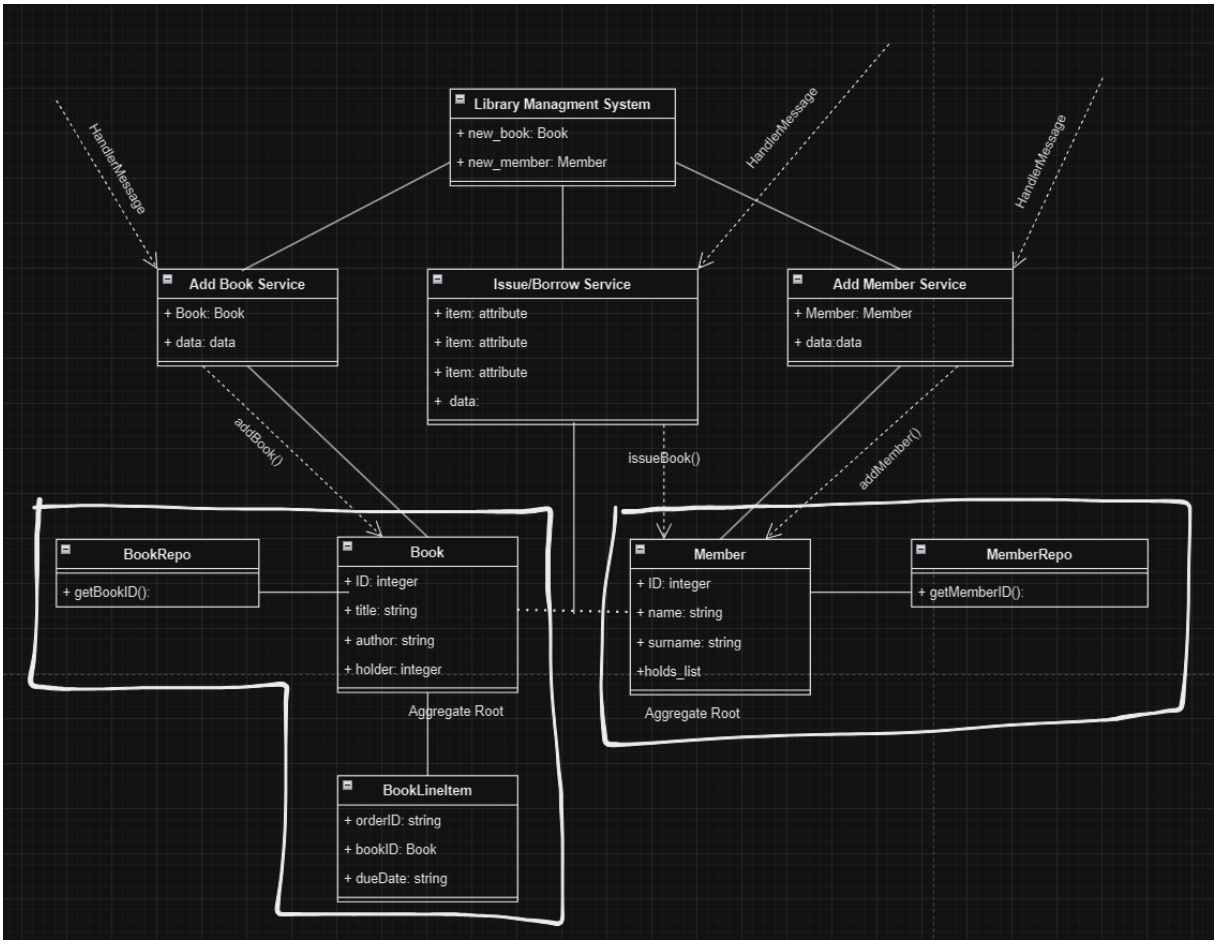
1- Defining the domain model of the use-cases



- Servislere gelen kesikli çizgiler messageBus'tan gelen eventlerdir. Bunlar root'lara ulaştırılır.

Library Management System

- Library Management System sistem yetkilerini alt servislere devreden kök domain servisimiz.
- Yetkileri devrettiği servisler entity üzerine gelen event'leri fonksiyonel olarak tutar.
    - Add Book Service
    - Add Member Service
    - Issue/Borrow Service

Book ve Member

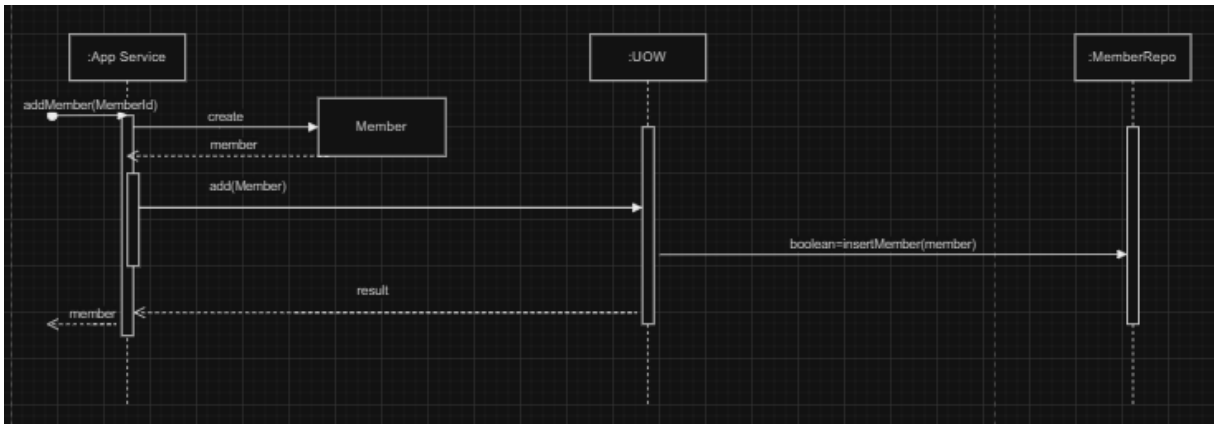
- Ayrı ayrı aggregate'lerde state değiştikçe yeni varlık ve entity oluşturduğu için entity'dir.
- Aynı zamanda bunlar ID'lere sahip yapılar bu yüzden entity olurlar. Repository'leri bu entitylerin ID'lerini tutan database bileşenleridir.
- AddMember yeni üye eklemedeki yönetimi sağlayan yapıdır.
- IssueBook, Book ve Member arasında geçici olarak oluşturlan ilişkiyi yöneten yapıdır.
- AddBook yeni kitap eklemede yönetimi sağlayan yapıdır.

Aggregate'ler

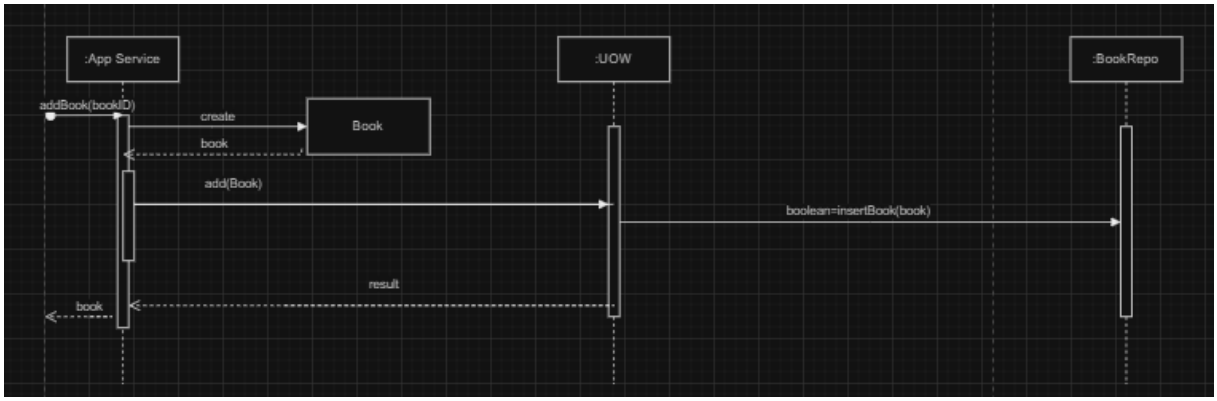
- Book, BookLineItem ve BookRepo aggregate'inin Root'u Book entity'si. BookLineItem ise Value Object.
- Member ve MemberRepo aggregate'inin Root'u ise Member entity'si.

2- Drawing the Sequence Diagrams of the use-cases and discussing our responsibility assignment decisions using the GRASP Patterns

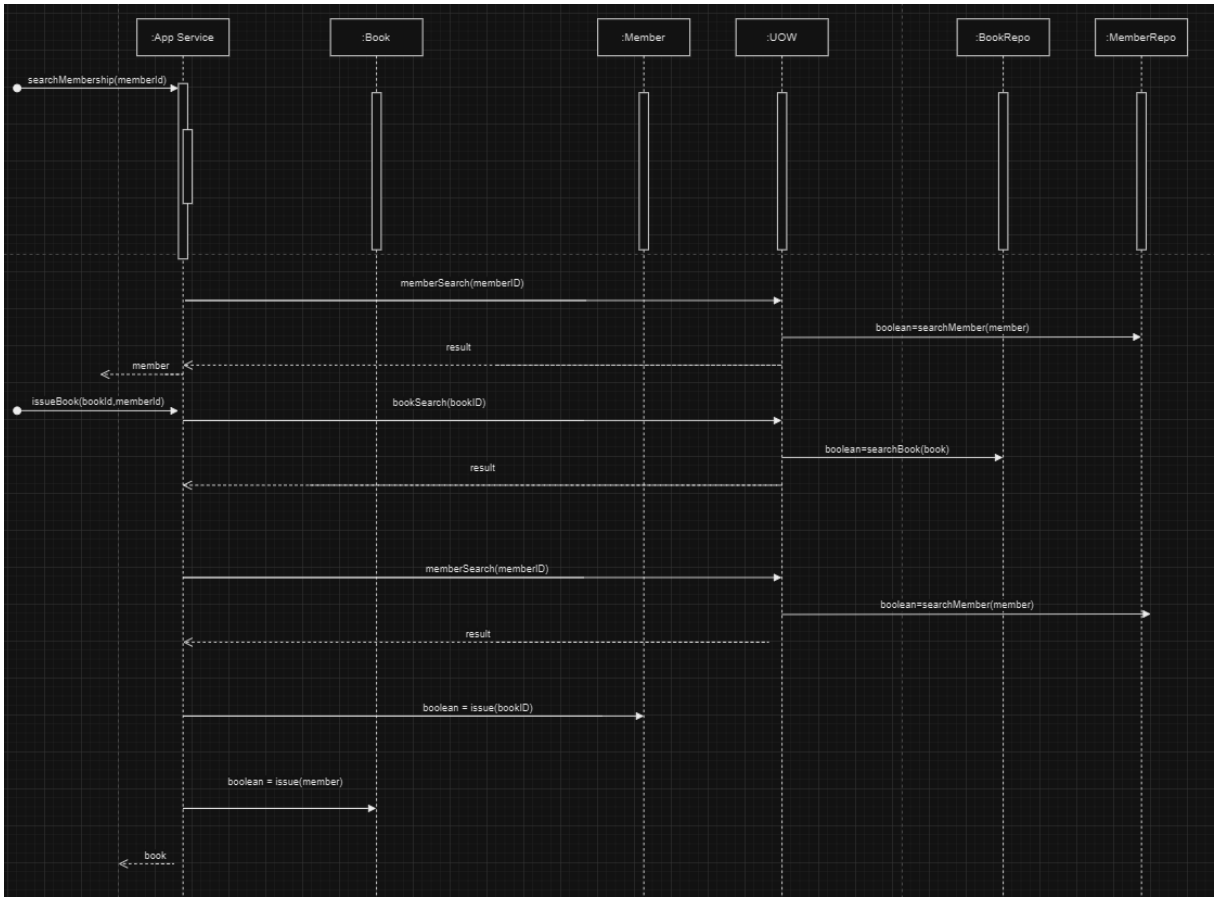
- Add Member Sequence Diagram



- Add Book Sequence Diagram



- Issue/Borrow Book Sequence Diagram



Library Management System

- Library management system servislere yetki veren ana yönetim sistemidir. Bu sebepten GRASP pattern'a göre **creator** olarak sayabiliriz.
- Benzer işlevlere sahip ve bir arada çalışabilen sistemleri içerir. Bu da sistemin modüler, düzenli ve işlevsel bir yapıya sahip olmasını sağlar. (**high cohesion**)

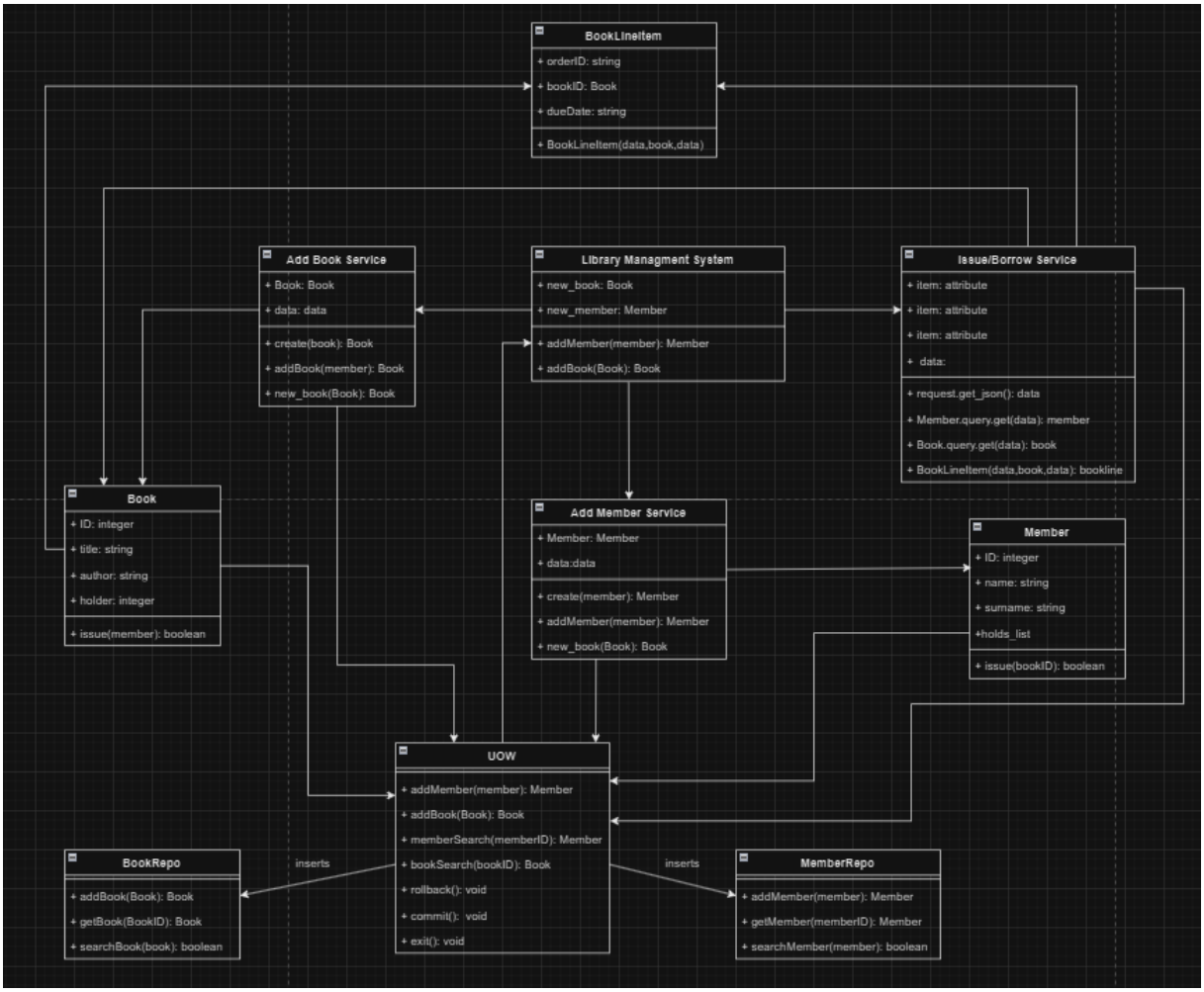
Book Line Item

- Book Line Item sadece veri taşıyan bir yapıdır ve veritabanı ve iş mantığı işlemleriyle doğrudan etkileşimde bulunmaz.
- Bir kitabın ödünç alınması ile ilgili sadece gereken bilgileri içerdiğinden bunu kullanmak ve yenisini oluşturmak diğer işlemlere minimum şekilde etki eder. Bu da sistemdeki esnekliği arttırdığından GRASP pattern'a göre **low coupling** sayılır.

Unit of Work (UoW)

- Veritabanı ve repository'nin görevlerini üstlenerek Flask'ın coupling'ini azaltır. (**low coupling**)
- Veritabanı ve session bilgilerine sahip olduğu için **Information Expert**'tir.
- Veritabanı işlemleri ve iş mantığı koordinasyonu gibi yüksek seviyeli görevlerin yönetiminden sorumlu bir yapıdır. Bu da GRASP pattern'a göre **controller**'ın temel sorumluluklarına uygundur.

3- Drawing the Class Diagrams for the designated use-cases



4- Implementing the use-cases

```
from flask import Flask, request
from flask_sqlalchemy import SQLAlchemy
from dataclasses import dataclass
from sqlalchemy import Table, Column, Integer, String, MetaData
from abc import ABC, abstractmethod
from typing import List

# Create Flask app instance and configure database
app = Flask(__name__)
app.config['test'] = 'test'  # Example database URI
db = SQLAlchemy(app)
metadata = MetaData()

# Create a new value obj for each book request
@dataclass(frozen=True)
class BookLineItem:
    orderid: str
    bookid:  Book
    duedate: str


# Define the Book model for the database
class Book(db.Model):
    __tablename__ = 'book'
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(25))
    author = db.Column(db.String(25))
    holder = db.Column(db.Integer)

    def __init__(self, title, author, bookid):
        self.title = title
        self.author = author
        self.id = bookid
        self.holder = None

# Define the Member model for the database
class Member(db.Model):
    __tablename__ = 'member'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(25))
    surname = db.Column(db.String(25))
    hold_list = db.Column(db.PickleType)

    def __init__(self, name, surname, memberid):
        self.name = name
        self.surname = surname
        self.id = memberid
        self.hold_list = []

    def allocate(self, line: BookLineItem):
        self.hold_list.append(line)

# Define the LibraryManagementSystem class
class LibraryManagementSystem:
    @staticmethod
    def add_member(uow, data):
        new_member = Member(data['name'], data['surname'],
```

```
data['memberid'])
        with uow:
            uow.session.add(new_member)
            uow.session.commit()
            MessageBus.publish(MemberAdded(new_member.id, new_member.name,
new_member.surname))

    @staticmethod
    def add_book(uow, data):
        new_book = Book(data['title'], data['author'], data['bookid'])
        with uow:
            uow.session.add(new_book)
            uow.session.commit()
            MessageBus.publish(BookAdded(new_book.id, new_book.title,
new_book.author))

    @staticmethod
    def issue_book(uow, data):
        member = Member.query.get(data['memberid'])
        book = Book.query.get(data['bookid'])
        bookline = BookLineItem(data['orderid'], book, data['duedate'])
        with uow:
            uow.issue_book(member, book, bookline)
            uow.session.commit()
            MessageBus.publish(BookIssued(member.id, book.id,
data['orderid'], data['duedate']))

# Routes
@app.route('/add_member', methods=['POST'])
def add_member_route(uow: UOW):
    data = request.get_json()
    LibraryManagementSystem.add_member(uow, data)
    return {"message": "New member added."}

@app.route('/add_book', methods=['POST'])
def add_book_route(uow: UOW):
    data = request.get_json()
    LibraryManagementSystem.add_book(uow, data)
    return {"message": "New book added."}

@app.route('/issue_book', methods=['POST'])
def issue_book_route(uow: UOW):
    data = request.get_json()
    LibraryManagementSystem.issue_book(uow, data)
    return {"message": "Book issued."}



class Repository:
    def __init__(self, session):
        self.session = session

    def add(self, entity):
        self.session.add(entity)

    def get(self, entity_class, entity_id):
        return self.session.query(entity_class).get(entity_id)

    def list(self, entity_class):
        return self.session.query(entity_class).all()
```

```
class BookRepository(Repository):
    def get_by_bookid(self, id):
        return self.session.query(Book).filter_by(id=id).first()

class MemberRepository(Repository):
    def get_by_member(self, id):
        return self.session.query(Member).filter_by(id=id).first()

class AbstractRepositoryMember(abc.ABC):
    @abc.abstractmethod
    def add(self, batch: Member.id):
        return NotImplementedError

    @abc.abstractmethod
    def get(self, reference) -> Member.id:
        return NotImplementedError

class AbstractRepositoryBook(abc.ABC):
    @abc.abstractmethod
    def add(self, batch: Book.id):
        return NotImplementedError

    @abc.abstractmethod
    def get(self, reference) -> Book.id:
        return NotImplementedError

class AbstractUnitOfWork(abc.ABC):
    batches: repository.AbstractRepository

    def __exit__(self, *args):
        self.rollback()

    @abc.abstractmethod
    def commit(self):
        return NotImplementedError

    @abc.abstractmethod
    def rollback(self):
        return NotImplementedError




class Event(ABC):
    pass

@dataclass(frozen=True)
class BookAdded(Event):
    book_id: int
    title: str
    author: str

@dataclass(frozen=True)
class MemberAdded(Event):
    member_id: int
    name: str
    surname: str

@dataclass(frozen=True)
```

```
class BookIssued(Event):
    member_id: int
    book_id: int
    order_id: str
    due_date: str

# Define the MessageBus class for event handling
class MessageBus:
    HANDLERS = {}

    @staticmethod
    def register_handler(event_type, handler):
        if event_type not in MessageBus.HANDLERS:
            MessageBus.HANDLERS[event_type] = []
        MessageBus.HANDLERS[event_type].append(handler)

    @staticmethod
    def publish(event):
        event_type = type(event)
        if event_type in MessageBus.HANDLERS:
            for handler in MessageBus.HANDLERS[event_type]:
                handler(event)


# Run the Flask app and create the database tables
if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```