Prac 1

Design principles, analysis and architectural patterns

Alejandro Pérez Bueno

Apr 30, 2024

Table of Contents

Question 1

a)

The **Composite Object** pattern is the ideal choice for representing the information due to its ability to treat both individual elements and collections of elements uniformly. In this scenario, object classes are the individual elements, while packages represent the collections. The pattern's key strength lies in its creation of a superclass that encompasses both elements and collections. This allows the rest of the system to remain agnostic to the specific subclass it interacts with. This feature proves particularly valuable in representing the hierarchical structure of packages containing other packages or object classes.

Basically, the Composite Object pattern enables a flexible and efficient representation of the hierarchical structure, where packages can contain other packages or individual object classes.

b)

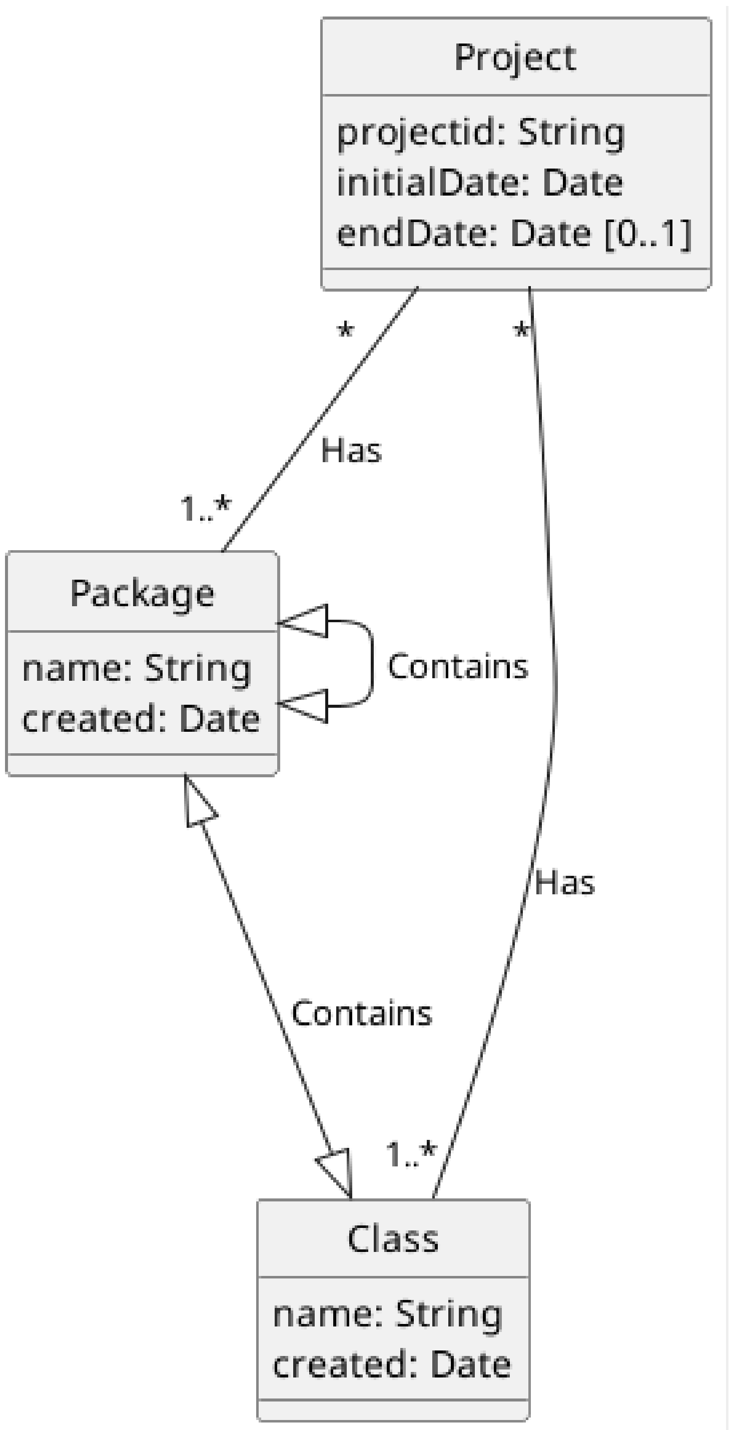

Figure 1: Static Analysis Diagram

Attributes:

```
Package:
    name: String
    created: Date
Class:
    name: String
    created: Date
```

Association Multiplicities:

```
Contains:
    Package to Package: 0..*
    Package to Class: 0..*
Has:
    Project to Package: 1..*
    Project to Class: 1..*
```

Integrity Constraints:

```
A package cannot contain itself either directly or indirectly.
```

Question 2

a)

The appropriate analysis pattern to represent this information is **Historical Association**. This pattern allows us to recover the values that an association has had in the past, which is necessary in this case where we want to store all developers who have been assigned to bugs and the period during which they were assigned.

b)

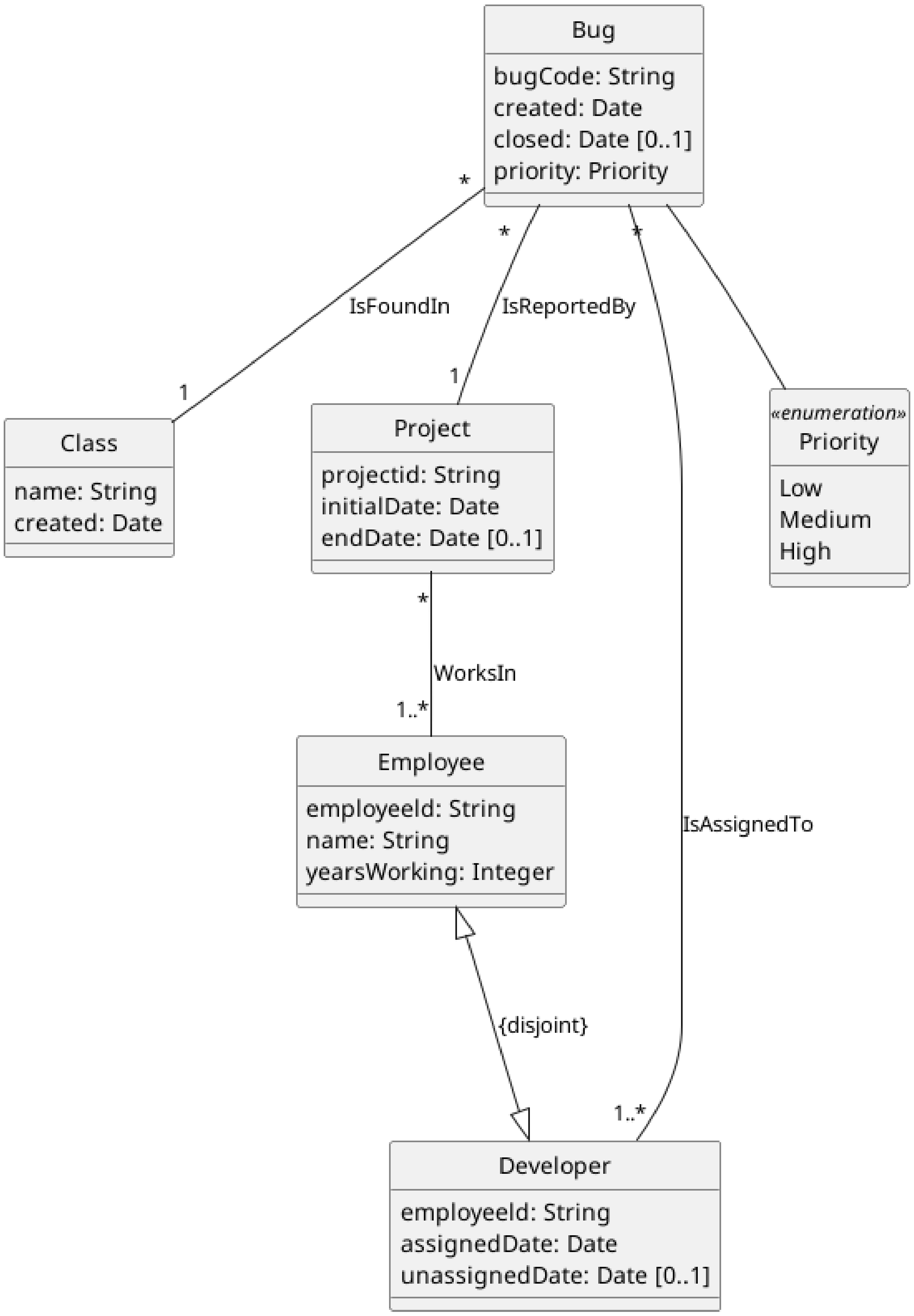

Figure 2: Static Analysis Diagram

Attributes:

```
Developer:
    assignedDate: Date
    unassignedDate: Date [0..1]
```

Association Multiplicities:

```
IsAssignedTo:
    Bug to Developer: 1..*
```

Integrity Constraints:

`The unassignedDate of a developer must be after the assignedDate.`

c)

Pseudocode:

```
def priorityBugs(self) -> Set[Tuple[str, str, str]]:
    result = set()
    for assignment in self.assignments:
      bug = assignment.bug  # Accessing the associated Bug object
      if bug.priority == Priority.High:
        bug_id = (bug.projectId, bug.className, bug.bugCode)
        result.add(bug_id)
    return result
```

Question 3

a)

The principle of interface segregation is not satisfied, because the DataBase class offers operations for each of the classes in the static analysis diagram. This means that clients of the DataBase class must depend on operations that they do not use. For example, a client that only needs to store and modify projects does not need to depend on the operations for saving and modifying classes and bugs.

b)

The principle of inversion of dependencies is not satisfied, because high level modules or classes should not depend on low level ones, but rather on an abstraction. In the diagram the Project class, which can be considered a high-level class, is directly dependent on the DataBase class, a lower-level class, for operations like saveProject and modifyProject. This creates a direct dependency from the high level class to the low-level class, which goes against the DIP.

Question 4

a)

Interface Segregation Principle (ISP):

- Create distinct interfaces for specific operation types.
- For instance, define ‘IProjectDatabaseOperations’ with ‘saveProject’ and ‘modifyProject’ methods, ‘IClassDatabaseOperations’ with ‘saveClass’ and ‘modifyClass’ methods, and so on.
- Client classes will only depend on the interfaces relevant to their operations.

Dependency Inversion Principle (DIP):

- Introduce an abstraction layer between high-level classes (e.g., Project, Class, Bug) and the low-level DataBase class.
- Create interfaces (as described above) and have high-level classes depend on these interfaces instead of the concrete DataBase class.
- The DataBase class will implement the defined interfaces.

b)

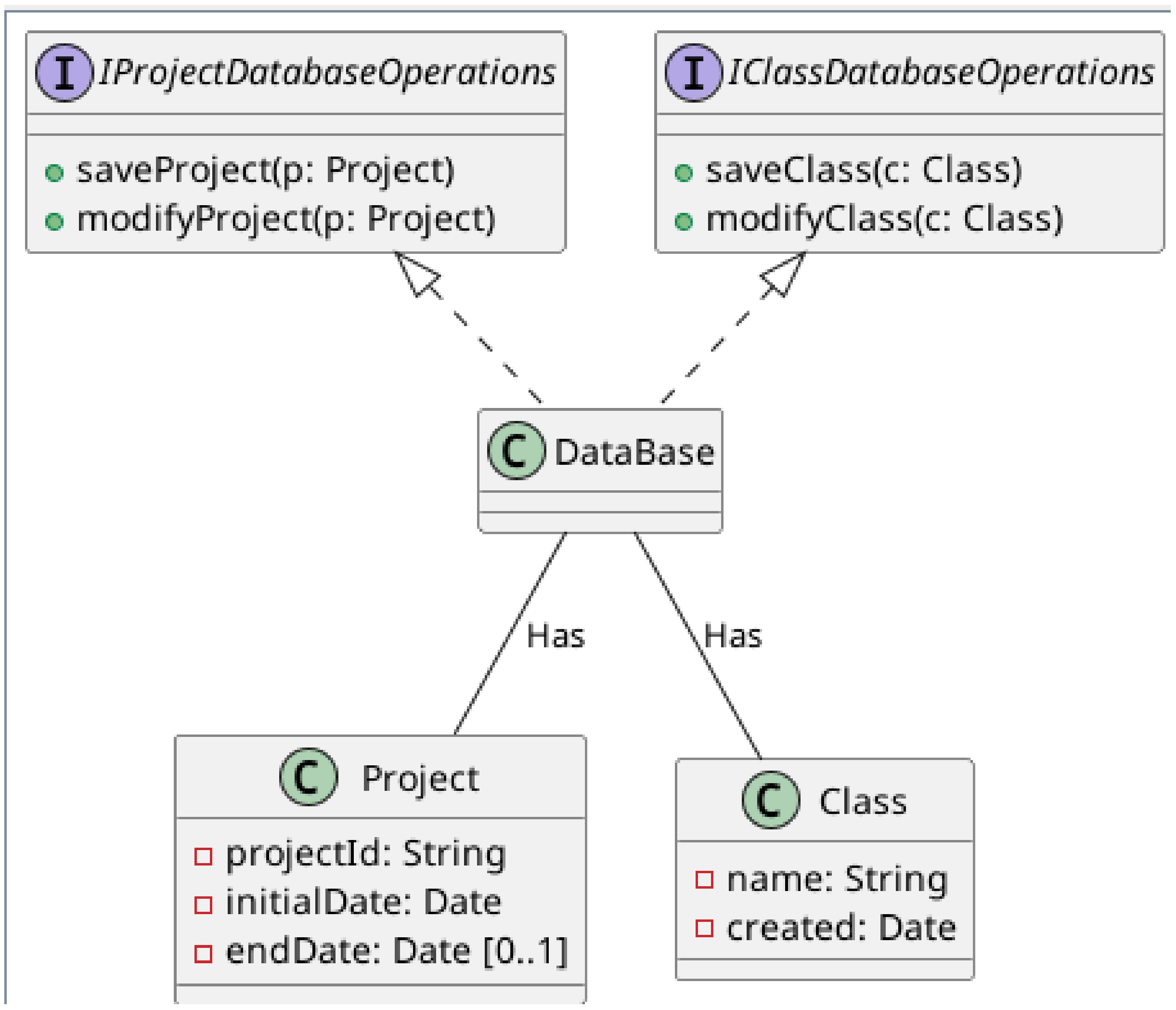

Figure 3: Static Analysis Diagram

Question 5

a)

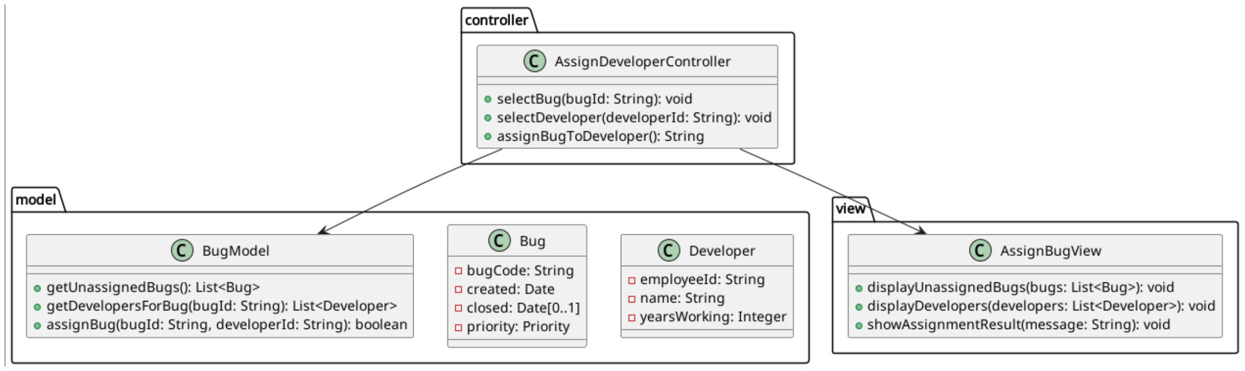
controller
AssignDeveloperController
selectBug(bugId: String): void
selectDeveloper(developerId: String): void
assignBugToDeveloper(): String
model
BugModel
getUnassignedBugs(): List<Bug>
getDevelopersForBug(bugId: String): List<Developer>
assignBug(bugId: String, developerId: String): boolean
Bug
bugCode: String
created: Date
closed: Date[0..1]
priority: Priority
Developer
employeeId: String
name: String
yearsWorking: Integer
view
AssignBugView
displayUnassignedBugs(bugs: List<Bug>): void
displayDevelopers(developers: List<Developer>): void
showAssignmentResult(message: String): void

Figure 4: Static Analysis Diagram

b)

```
# The view displays all unassigned bugs
unassignedBugs = BugModel.getUnassignedBugs()
AssignBugView.displayUnassignedBugs(unassignedBugs)

# Project manager selects a bug
selectedBugId = # get the selected bug ID from the view
AssignDeveloperController.selectBug(selectedBugId)

# The view displays all developers that can be assigned to the selected bug
availableDevelopers = BugModel.getDevelopersForBug(selectedBugId)
AssignBugView.displayDevelopers(availableDevelopers)

# Project manager selects a developer
selectedDeveloperId = # get the selected developer ID from the view
AssignDeveloperController.selectDeveloper(selectedDeveloperId)

# Assign the bug to the developer
assignmentSuccess = BugModel.assignBug(selectedBugId, selectedDeveloperId)

# Display the result message
if assignmentSuccess:
    AssignBugView.showAssignmentResult("Successfully assigned")
else:
    AssignBugView.showAssignmentResult("Assignment failed")
```

Question 6

a)

Technical Services Layer (Data Layer):

- Exercise 4:
  - DataBase: This class directly interacts with the data storage and retrieval mechanisms, clearly placing it in the data layer.
  - IProjectDatabaseOperations and IClassDatabaseOperations: These interfaces define operations for data access related to projects and classes, making them part of the data layer.
- Exercise 5:
  - Data Access Classes: While not explicitly shown, classes responsible for interacting with the database to retrieve and manipulate Bug, Developer, and related data would reside here.

The data layer houses classes and interfaces that handle data persistence, retrieval, and manipulation. These elements interact directly with databases, file systems, or other data sources.

b)

Domain Layer (Business Logic):

- Exercise 1:
  - Project, Package, and Class: These classes represent core business entities and their relationships within the software development domain.
- Exercise 2:
  - Bug, Project, Employee, Developer, and Priority: These classes represent entities and concepts central to the bug tracking and project management domain. They encapsulate business rules and logic.

The domain layer contains classes representing the core business entities, their attributes, and the relationships between them. It encapsulates the business logic and rules without concern for how data is stored or presented.

c)

Presentation Layer (User Interface):

- Exercise 5:
  - AssignBugView: This class is responsible for displaying information to the user and capturing user interactions related to bug assignment.

  - AssignDeveloperController: This class handles user input, interacts with the domain layer to process business logic, and updates the view accordingly.

The presentation layer consists of classes responsible for user interaction, including displaying information, capturing input, and mediating communication between the user and the domain layer.