Design Patterns

Project Description:

Teach-Me app that enables users(students) to take a test – topic wise. This app has skill to understand the user's ability and asks the questions accordingly, at the end of test, a report is generated, that gives a brief about the strengths and weakness of user for the particular topic.

An Admin kind of user can upload a test questionnaire using '.xls' document, having variety of questions with different difficulty level. Based on which, the students will be ranked or classified.

Group 30:

- Rahul Pandya (110024678) – pandya51@uwindsor.ca
- Aayushee Dave (110023928) – dave71@uwindsor.ca
- Richa Gupta (110013520) – gupta14h@uwindsor.ca
- Manan Parmar (110022360) – parmar6@uwindsor.ca

Note:

1. For examples with Diagram, documentation is listed in a note
2. For examples with Code, please read the content for documentation

References:

1. Class Slides
2. https://www.geeksforgeeks.org/

GRASP - General Responsibility Assignment Software Patterns

1. Creator

Problems and Constraints:

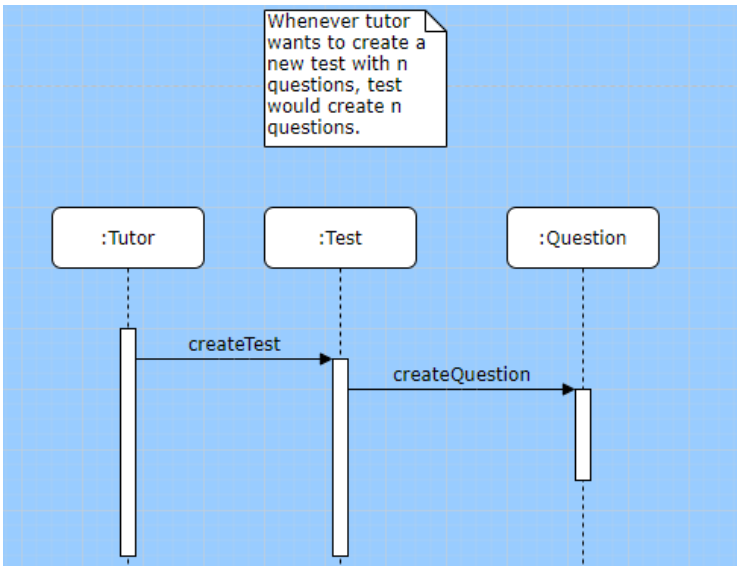
- Who creates an object?
- Who should create a new instance of a class?

Solution:

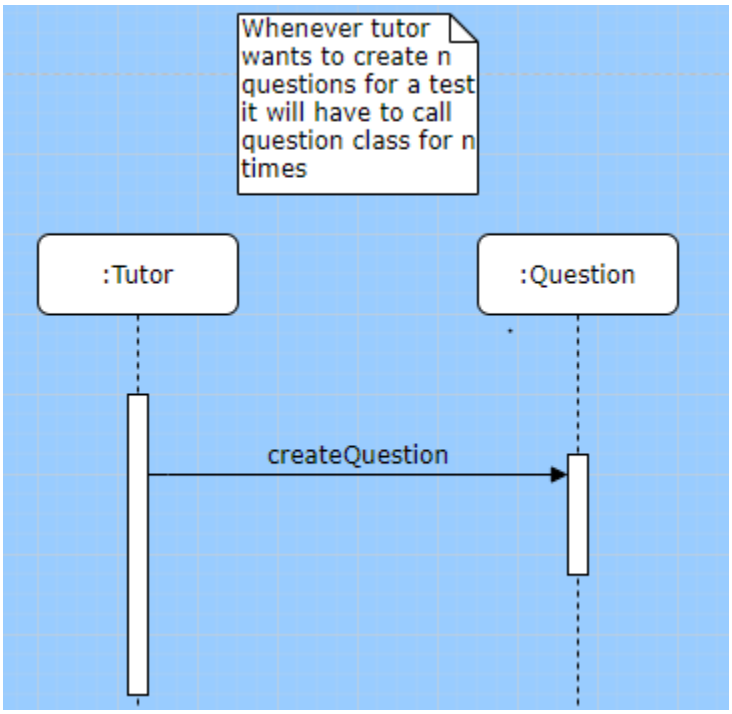
- Assign class B the responsibility to create object A if one of these is true (more is better)
  – B contains or compositely aggregates A
  – B records A
  – B closely uses A
  – B has the initializing data for A

Example:

- Good Design:



- Bad Design:


Whenever tutor wants to create n questions for a test it will have to call question class for n times
:Tutor
:Question
createQuestion

2. Controller

Problems and Constraints:

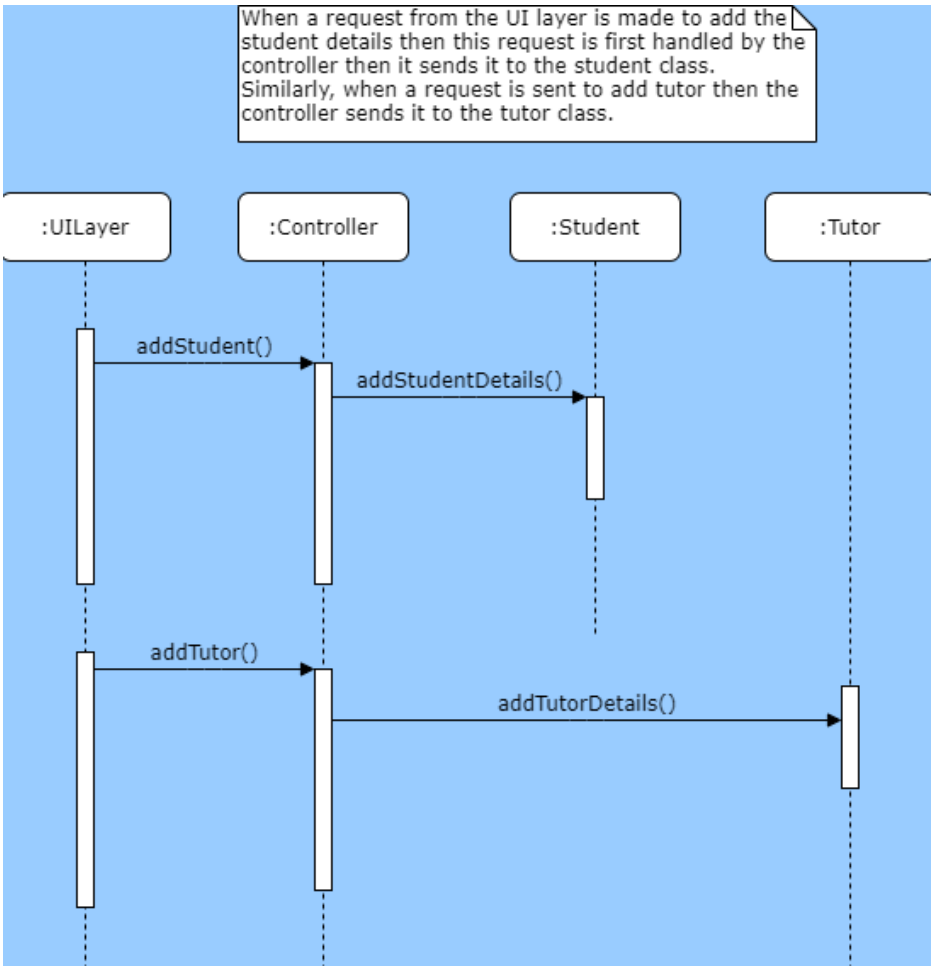
- Which object beyond UI Layer receives and coordinates the system operation?

Solution:

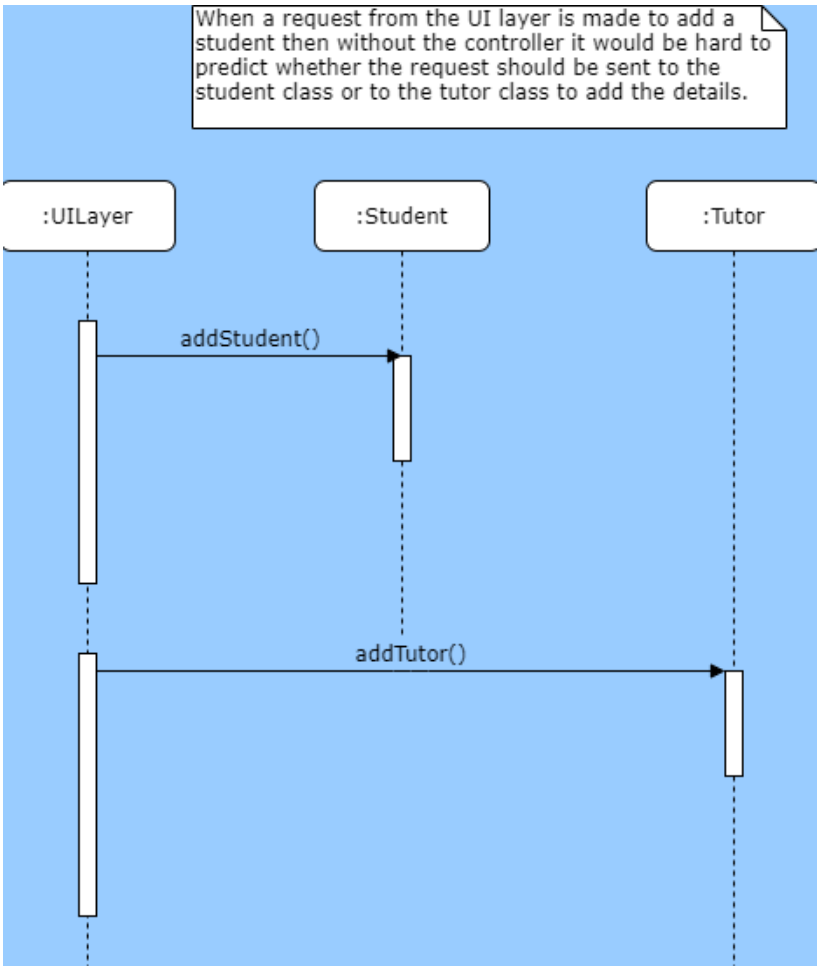
- Assign the responsibility to an object representing one of these choices:
  - Represents the overall “system”, “root object”, device that the software is running within, or a major subsystem.
  - Represents a use case scenario within which the system operation occurs.

Example:

- Good Design:



- Bad Design:


When a request from the UI layer is made to add a student then without the controller it would be hard to predict whether the request should be sent to the student class or to the tutor class to add the details.
:UILayer
:Student
:Tutor
addStudent()
addTutor()

3. Information Expert

Problems and Constraints:

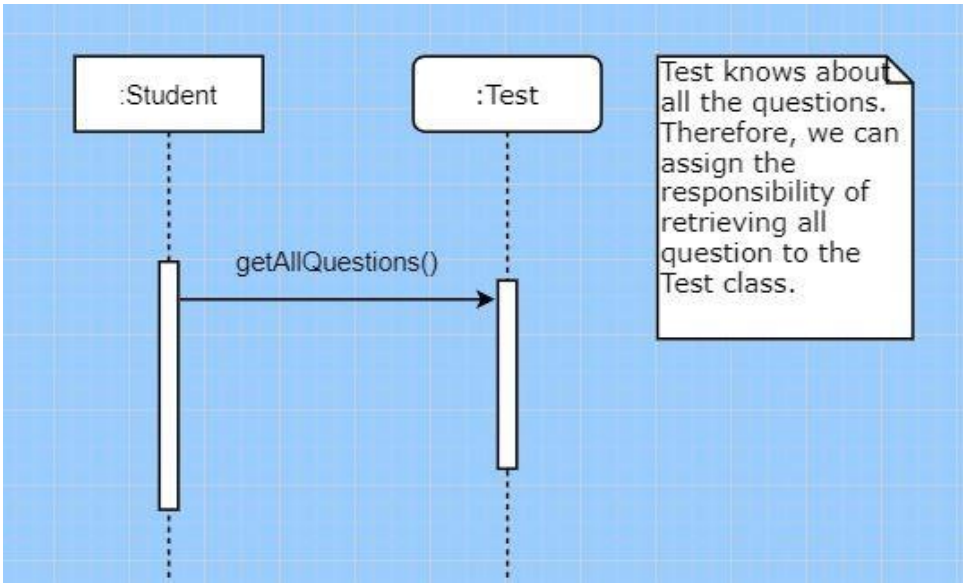
- Which responsibilities can be assigned to a particular object?

Solution:

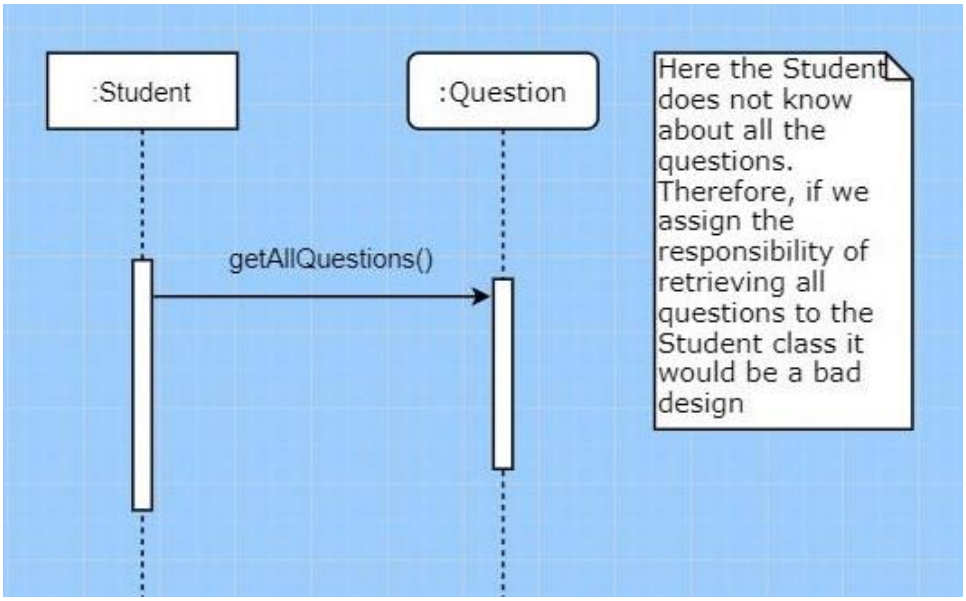
- Assign a responsibility to the class that has the information needed to fulfill it.

Example:

- Good Design



- Bad Design



4. Low Coupling

Problems and Constraints:

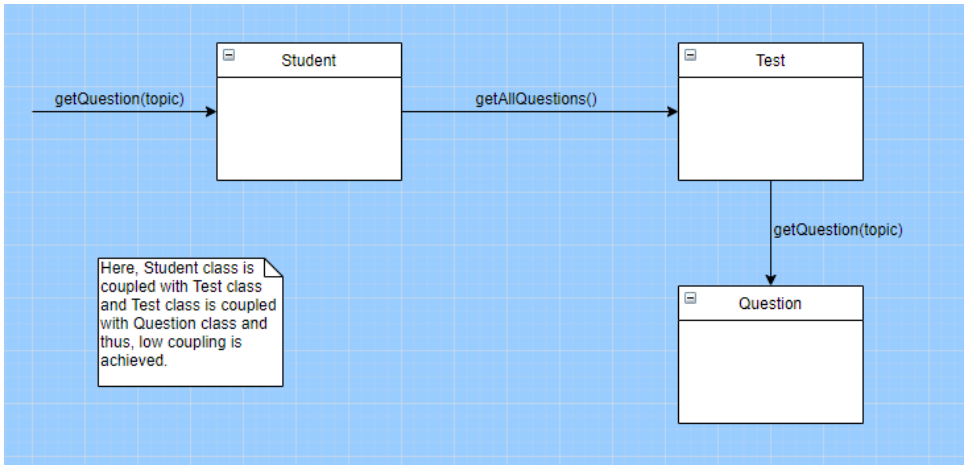
- How strongly the objects are connected to each other?
- How to reduce the impact of change?

Solution:

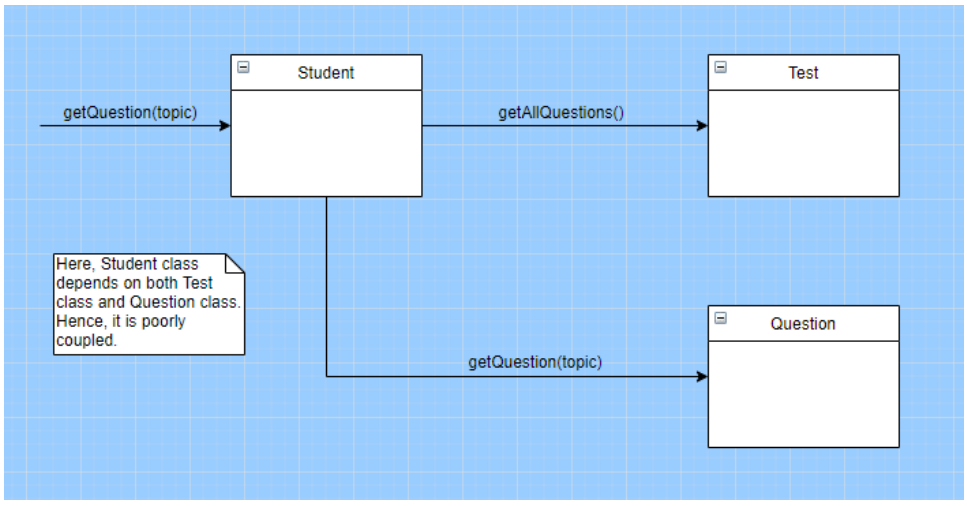
- Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

Example:

- Good Design



- Bad Design



5. High Cohesion

Problems and Constraints:

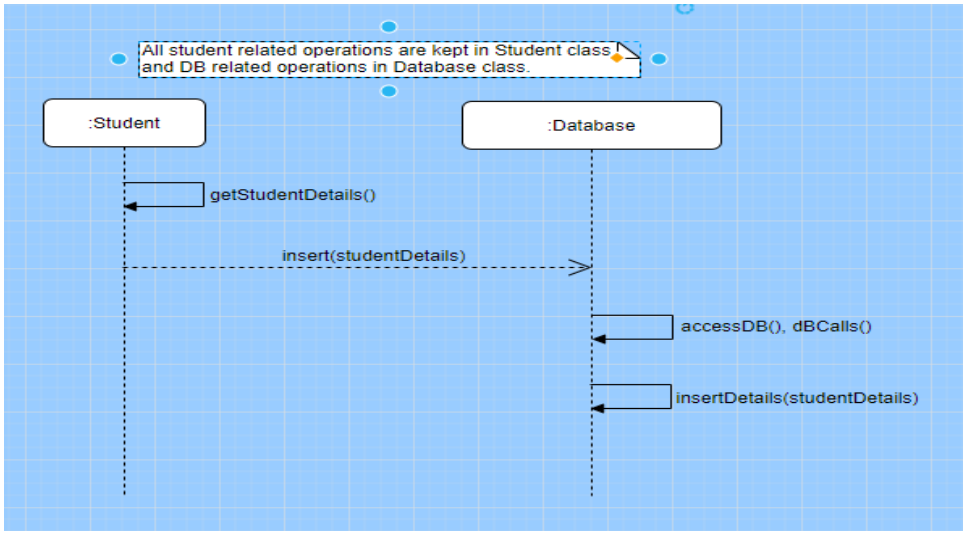
- How are the operations of any element functionally related?
- How to keep objects focused, understandable, manageable, and as a side effect, support Low Coupling?

Solution:

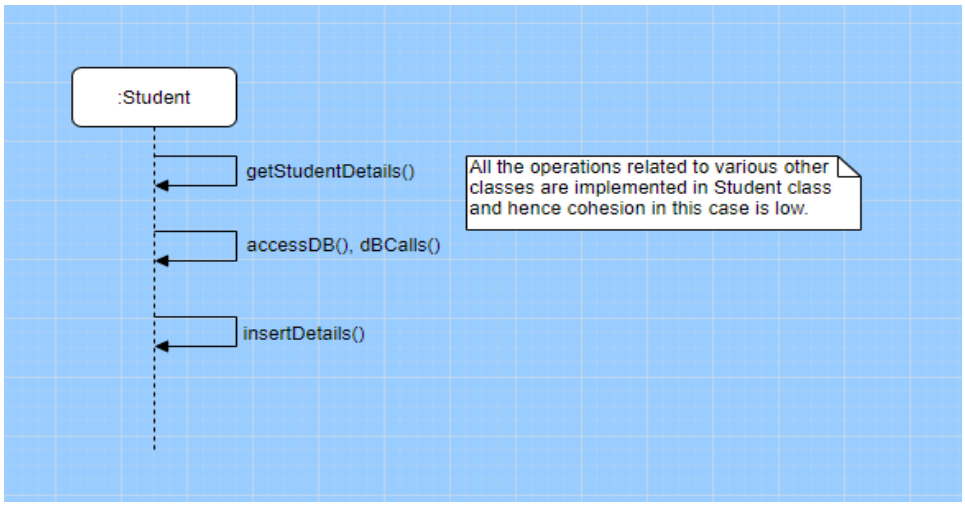
- Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

Example:

- Good Design



- Bad Design



6. Indirection

Problems and Constraints:

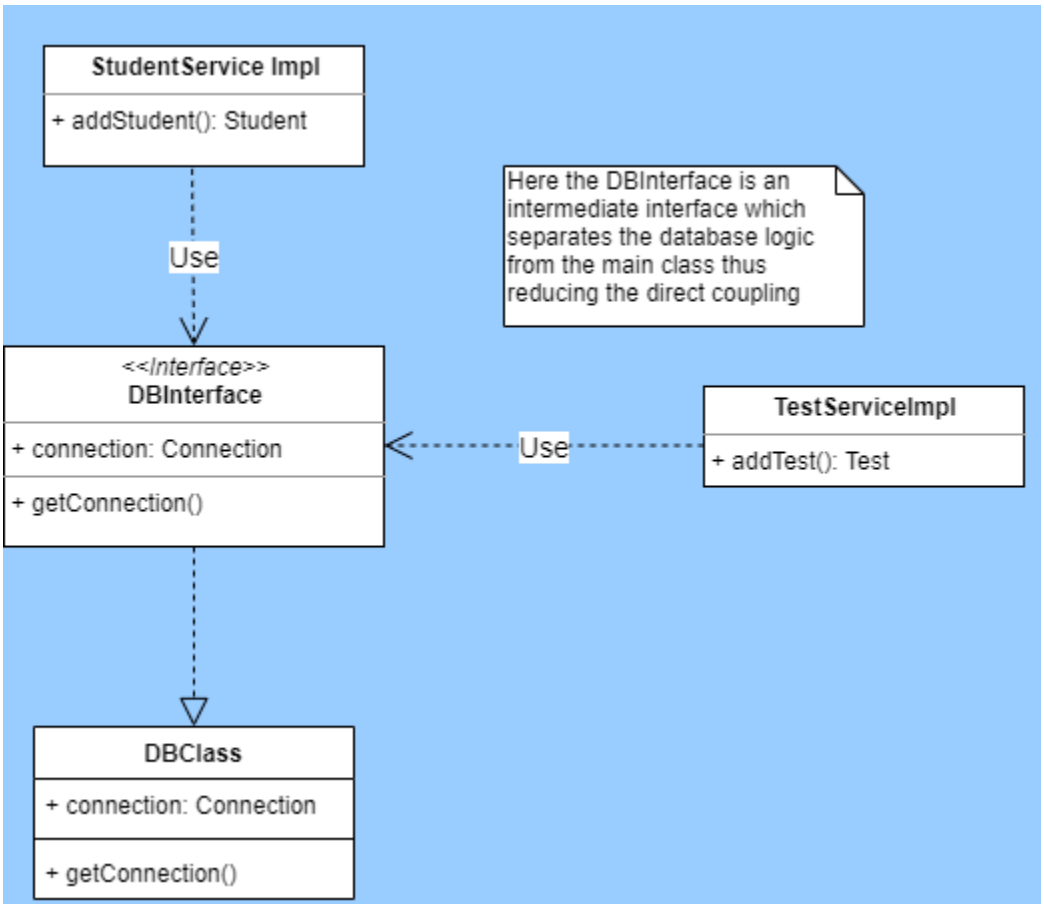
- How to assign responsibilities to avoid direct coupling between two or more elements?

Solution:

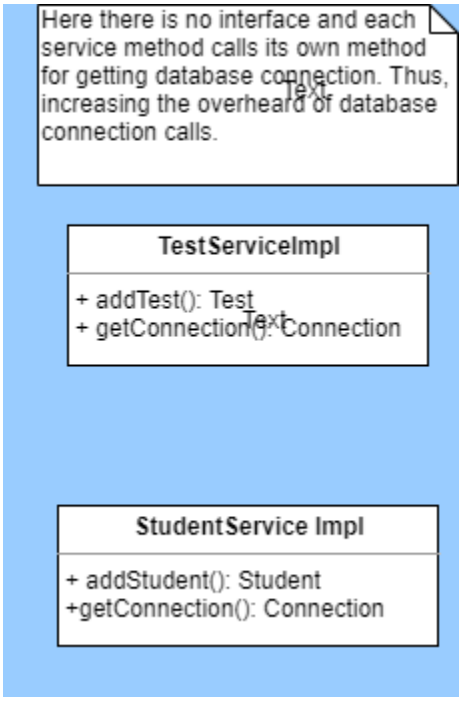
- Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

Example:

- Good Design



- Bad Design


Here there is no interface and each service method calls its own method for getting database connection. Thus, increasing the overheard of database connection calls.
TestServiceImpl
+ addTest(): Test
+ getConnection(): Connection
StudentService Impl
+ addStudent(): Student
+getConnection(): Connection

7. Polymorphism

Problems and Constraints:

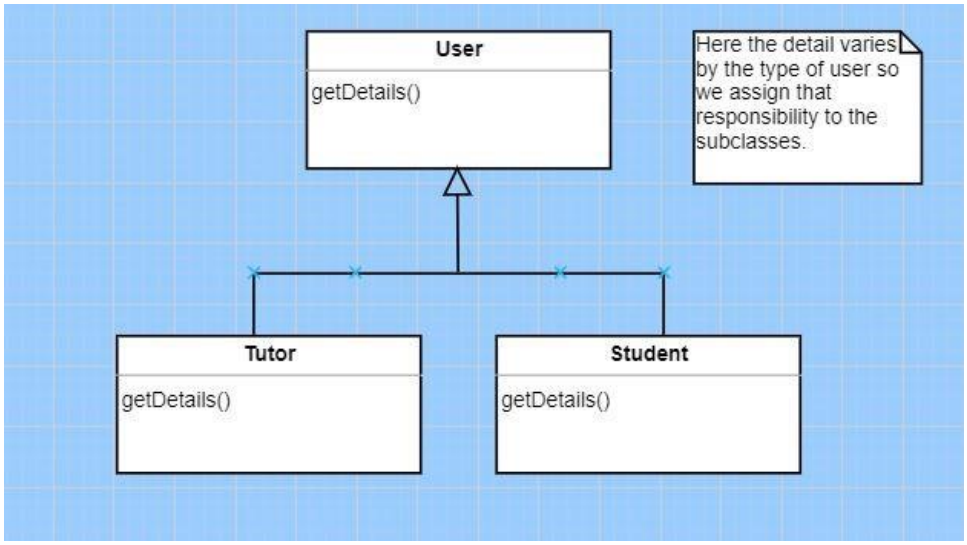
- Who should be responsible if the behavior changes by type?

Solution:

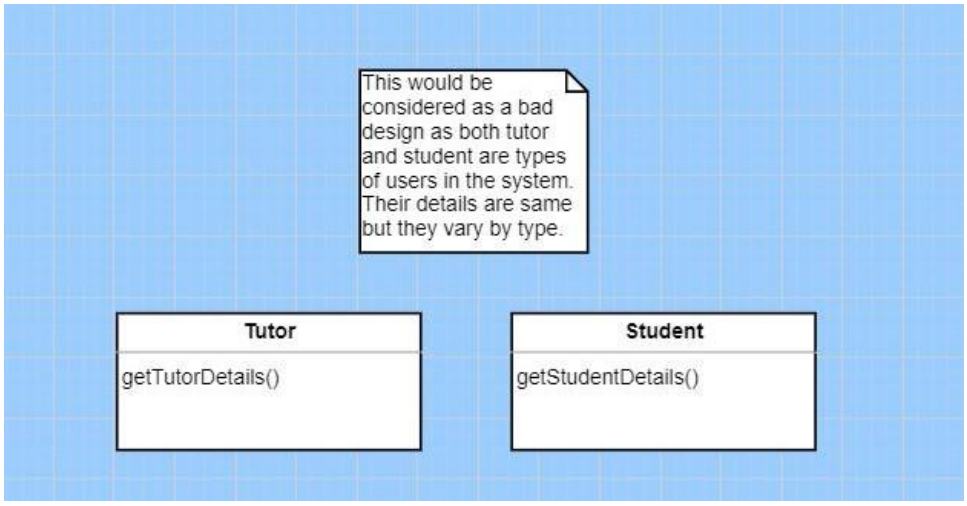
- When related alternatives or behaviors vary by type, assign responsibility to for the behavior- using polymorphic operations- to the types for which behavior varies.

Example:

- Good Design



- Bad Design



8. Protected Variations

Problems and Constraints:

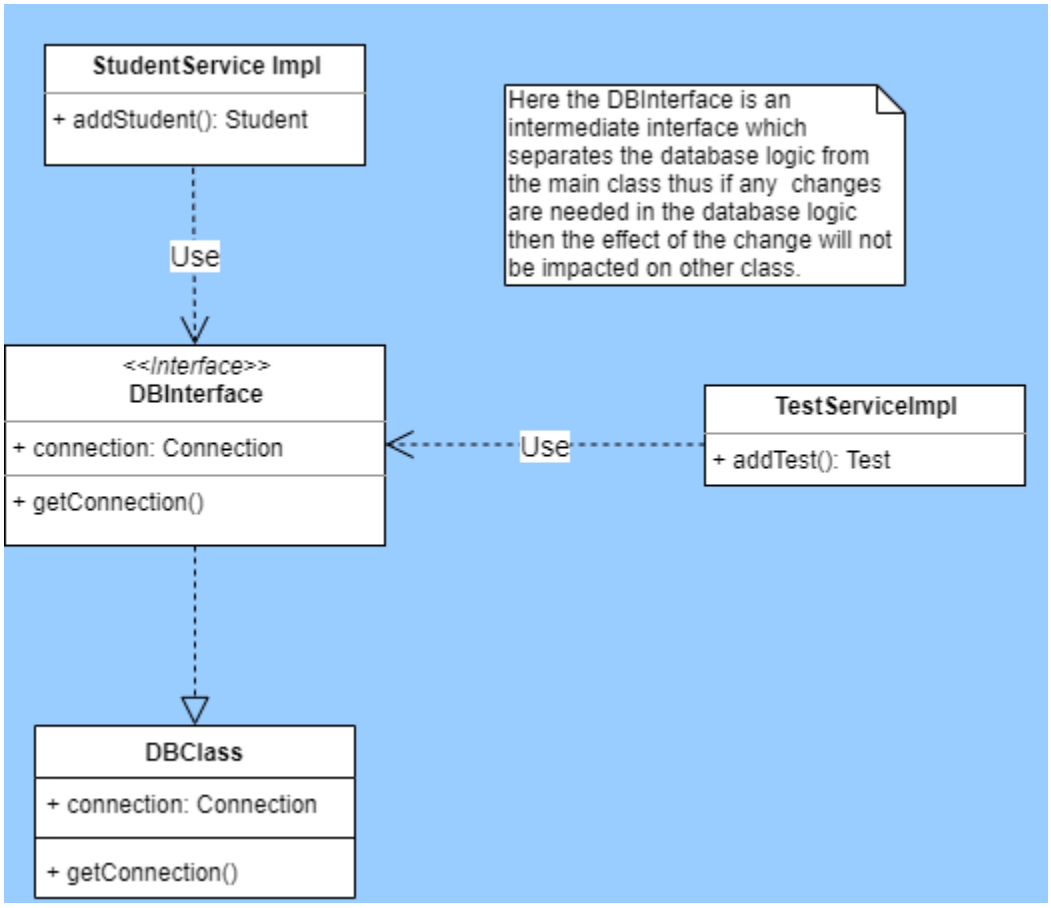
- How to avoid the impact of some variations of some elements on the other elements?

Solution:

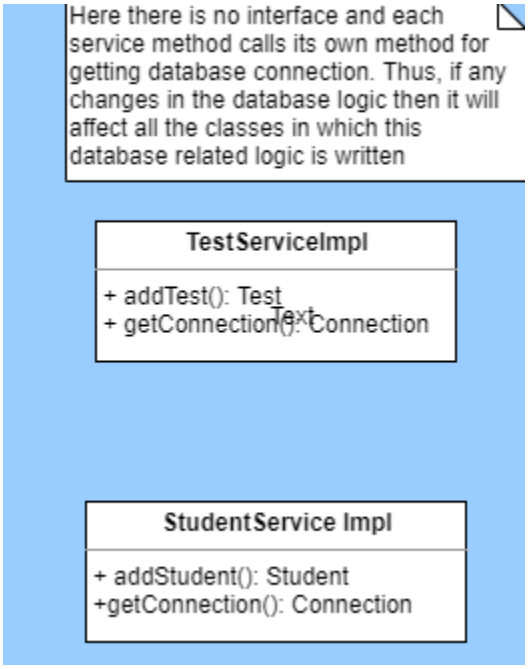
- Identify points of predicted variation or instability, assign responsibility to create a stable “interface” around them.

Example:

- Good Design



- Bad Design


Here there is no interface and each service method calls its own method for getting database connection. Thus, if any changes in the database logic then it will affect all the classes in which this database related logic is written
TestServiceImpl
+ addTest(): Test
+ getConnection(): Connection
StudentService Impl
+ addStudent(): Student
+getConnection(): Connection

9. Pure Fabrication

Problems and Constraints:

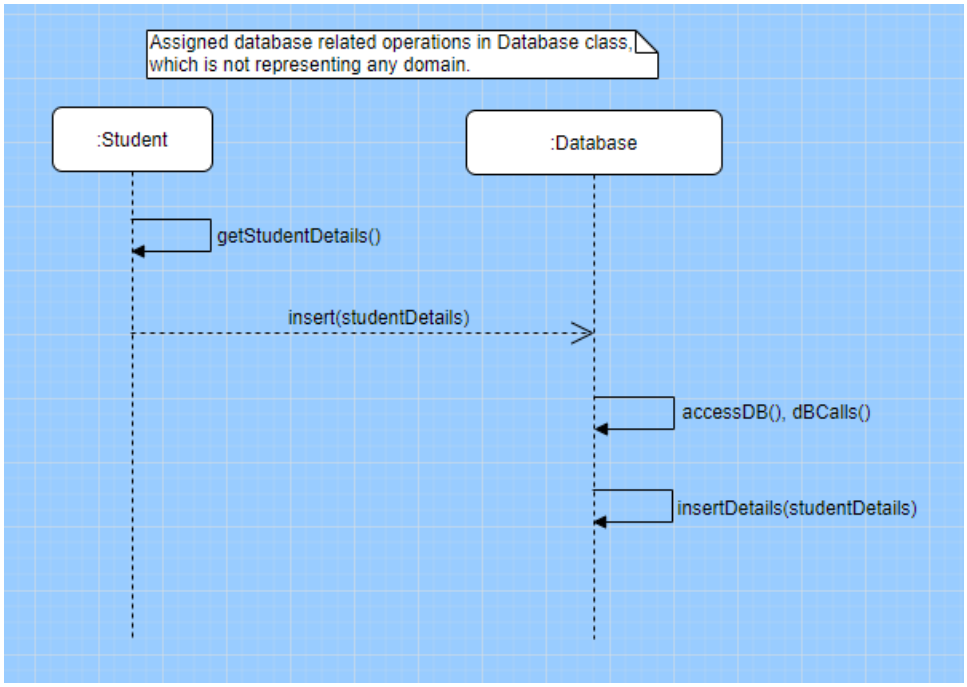
- What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling but solutions offered by other principles are not appropriate?

Solution:

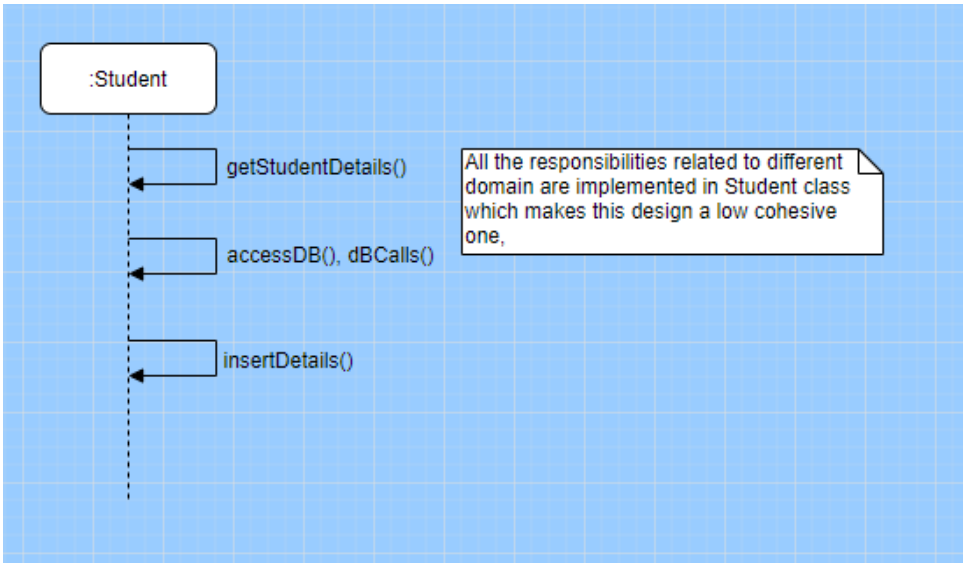
- Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept- something made up, in order to support high cohesion, low coupling and reuse.

Example:

- Good Design



- Bad Design


:Student
getStudentDetails()
accessDB(), dBCalls()
insertDetails()
All the responsibilities related to different domain are implemented in Student class which makes this design a low cohesive one,

GoF – Gang of Four

1. Singleton

Problems and Constraints:

- Ensure that a class has just a single instance.
- Provide a global access point to that instance.

Solution:

- In case of creating a singleton object, make the constructor of that class private, so that no other class can initialize the object with new keyword.
- In addition to that, we can use a static variable/object, that is shared by the whole class.

Example:

- **Good Design**:
  Creation of a common Database instance to use across all the modules of the project i.e. creating a class that returns a static connection object for DB

```
public class Connect {
...
    private static Connection conn = null;
...
    static public Connection getConnection() {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(DB_STR, USER, PASS);
        ...
        return conn;
    }
}
```

  Here, the Connection object: `conn` is shared across the whole application maintaining single connection to DB

- **Bad Design:**

  Creating a new instance for DB when required.

  Every time when we want to connect to DB, if we create a new connection object, we may end up with memory leaks.

2. Bridge

Problems and Constraints:

- How to separate the abstraction from the implementation of any method?

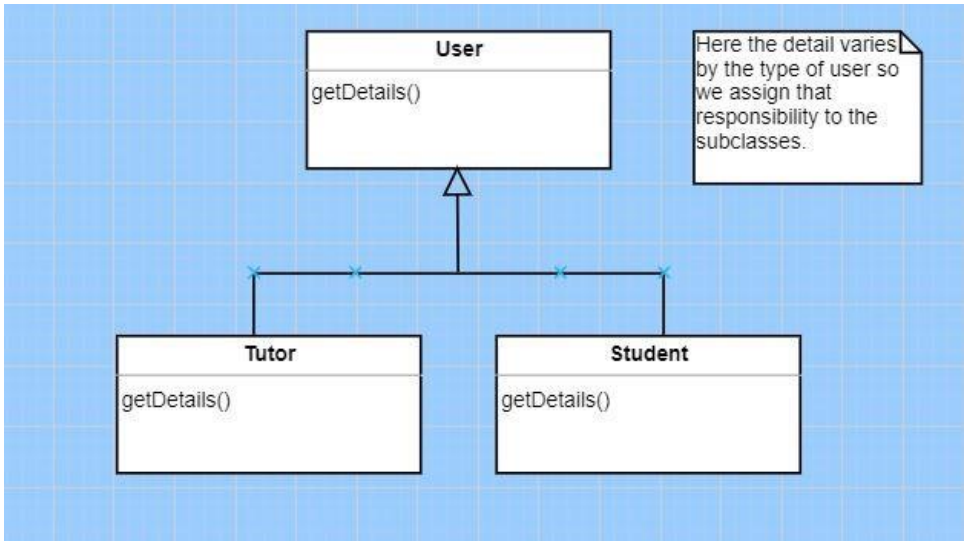
Solution:

- Create an Abstract Class.
- Extend that class to a class so that we can override and implement the method into that class.

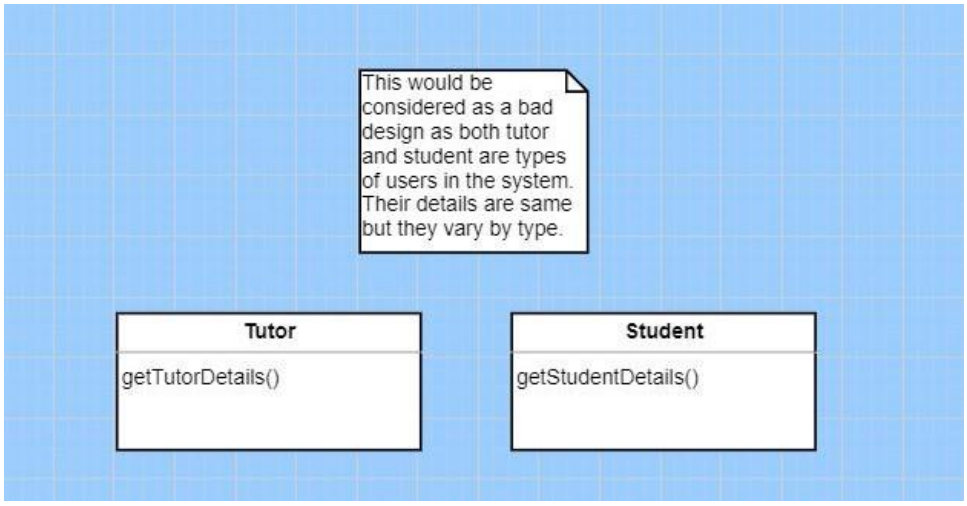
Example:

Behavior of Bridge is also explained by Polymorphism Pattern by GRASP.

- Good Design



- Bad Design



3. Builder

Problems and Constraints:

- To create or construct complex object and maintaining readability and reusability of code at the same time.

Solution:

- Create a Builder Class that provides a level of abstraction to create the Complex class.
- The Architect class can use the function(setters) of the builder class to set the instance variables of complex class and the builder method can call the constructor at the end to call the constructor of complex class and return the object of the same.

Example:

- **Good Design**:

  Complex Class: User (note that in builder design, the constructor od complex class should be private)

```
public class User {
    private long id;
    private String name;
    private UserType userType;
    private String email;
    private long contactNumber;

    private User(long id, String name, UserType userType, String email,
long contactNumber) {
        this.id = id;
        this.name = name;
        this.userType = userType;
        this.email = email;
        this.contactNumber = contactNumber;
    }
}
```

  Builder Class: UserBuilder (can be inside or outside the Complex Class or it can be the same User class too)

```
public static class UserBuilder {
    private long id;
    private String name;
    private UserType userType;
    private String email;
    private long contactNumber;

    public UserBuilder setId(long id) {
        this.id = id;
        return this;
    }

    public UserBuilder setName(String name) {
        this.name = name;
        return this;
```

```
    }

    public UserBuilder setUserType(UserType userType) {
        this.userType = userType;
        return this;
    }

    public UserBuilder setEmail(String email) {
        this.email = email;
        return this;
    }

    public UserBuilder setContactNumber(long contactNumber) {
        this.contactNumber = contactNumber;
        return this;
    }

    public User build() {
        return new User(id, name, userType, email, contactNumber);
    }
}
```

Architect Class: main Class (this can be the class that wants to construct User object)

…

```
User user = new temp.User.UserBuilder()
        .setId(1)
        .setContactNumber(1234567890)
        .setEmail("abcxyz@gmail.com")
        .setName("John Doe")
        .setUserType(UserType.TUTOR)
        .build();
```

…

- **Bad Design**:

  Using a Constructor having multiple parameters like

  ```
  User user = new User(1, "John Doe", UserType.TUTOR,
  "abcxyz@gmail.com", 1234657890);
  ```

  This can lead us to problems where we miss some of the parameters or interchange some of the values.

  And if sometimes we want to create an object with only half of the parameters then we have to create many constructors in that regards while builder pattern can save that time to code and allow users to skip some of the parameters leaving them to default ‘null’ or ‘0’ values.

4. Factory Method

Problems and Constraints:

- A reusable method that is responsible to create objects without specifying the exact class of the object that will be created

Solution:

- Create a static method that calls the constructor or initializer in sub class or same class.

Example:

- **Good Design:**

  Consider the Class User which is responsible to create a User whenever required.

  We can create a factory method that is static and can call the constructor to create new User.

  i.e.:

```
public class User {
    private long id;
    private String name;
    private UserType userType;
    private String email;
    private long contactNumber;

     private User(long id, String name, UserType userType, String email,
long contactNumber) {
        this.id = id;
        this.name = name;
        this.userType = userType;
        this.email = email;
        this.contactNumber = contactNumber;
    }

    public static User createUser(long id, String name, UserType
userType, String email, long contactNumber) {
        return new User(id, name, userType, email, contactNumber);
    }
}
```

  In main Class we just need to call that static method to create a user

  i.e.:

```
User user = User.createUser(1, "John Doe", UserType.TUTOR,
"abcxyz@gmail.com", 1234657890);
```

- **Bad Design:**

  A bad design can be calling the constructor every time we need to create the Object.

  The conventional way occupies memory, as this factory method is static, we can call this method whenever or wherever we require in the application.

5. Iterator

Problems and Constraints:

- Is there a way to go through each element of the any collection without accessing the same elements over and over?

Solution:

- The solution is to traverse to extract the traversal behavior of collection into a separate object called an iterator.

Example:

- **Good Design:**

  Using Iterator for iterating through list.

  i.e.

```
Iterator iterator = questionList.createIterator();
// iterating through all the question, irrespective of the count
while (iterator.hasNext())
{
   Question question = iterator.next());
   // do operation
}
```

  Here, iterator doesn't care if questionList is an Array or ArrayList or Tree or anything. It just iterates over the number of questions.

- **Bad Design:**

  We can iterate through anything using a conventional 'for();' loop, but if we were to use an Array of Question then, we have to use '`questionList.length`' to iterate

```
for(int i = 0; i < questionList.length; i++) {
   //  do operation
}
```

  and if it were an Array List, then we have to use '`questionList.size()`'

```
for(int i = 0; i < questionList.size(); i++) {
   //  do operation
}
```

  Iterator can solve this issue.