1 Class Diagrams Question

Please read this question carefully. Both the text and the diagrams contain essential information needed to complete the goals of this question.

1.1 Scenario

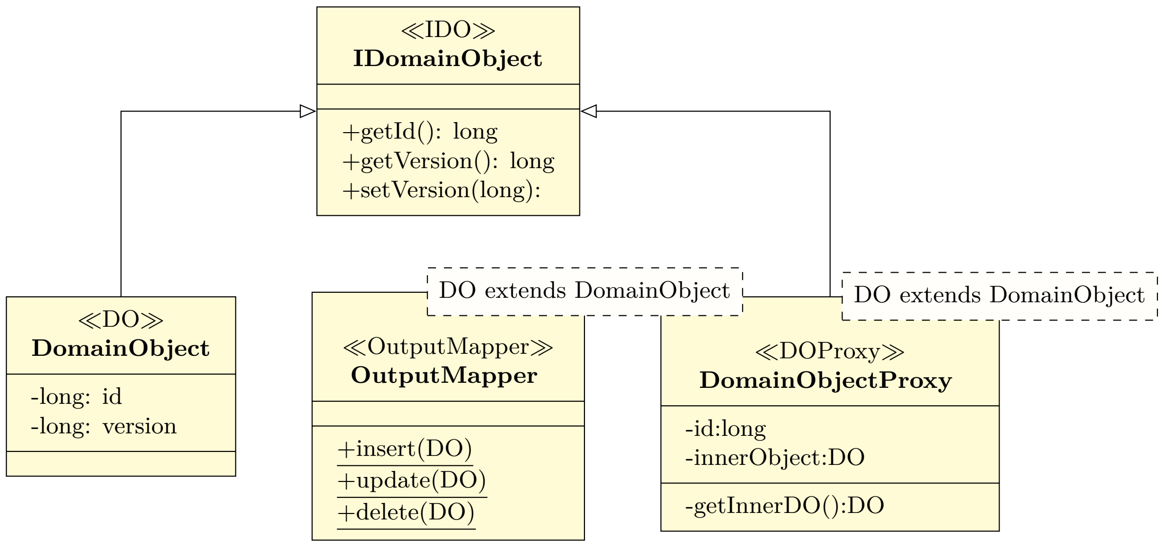
In this question and the next, you are tasked with developing a Pokemon card game online. You have been provided with some guidance as to the architectural structures to use. In addition, you have been given a breakdown of how some existing Domain Objects work together for this project. In this question, you will need to show how the Game Domain Object is represented across all the architectural patterns required, and how all these classes relate to the classes involved in the View Board Use Case from Assignment 2. This diagram should be broken down by layer.

1.2 Additional Guidance

Some existing classes are provided with the framework being used in this project. For the sake of clarity, a brief description of some of these classes are provided, along with an indication on how they can be used to simplify your diagram.

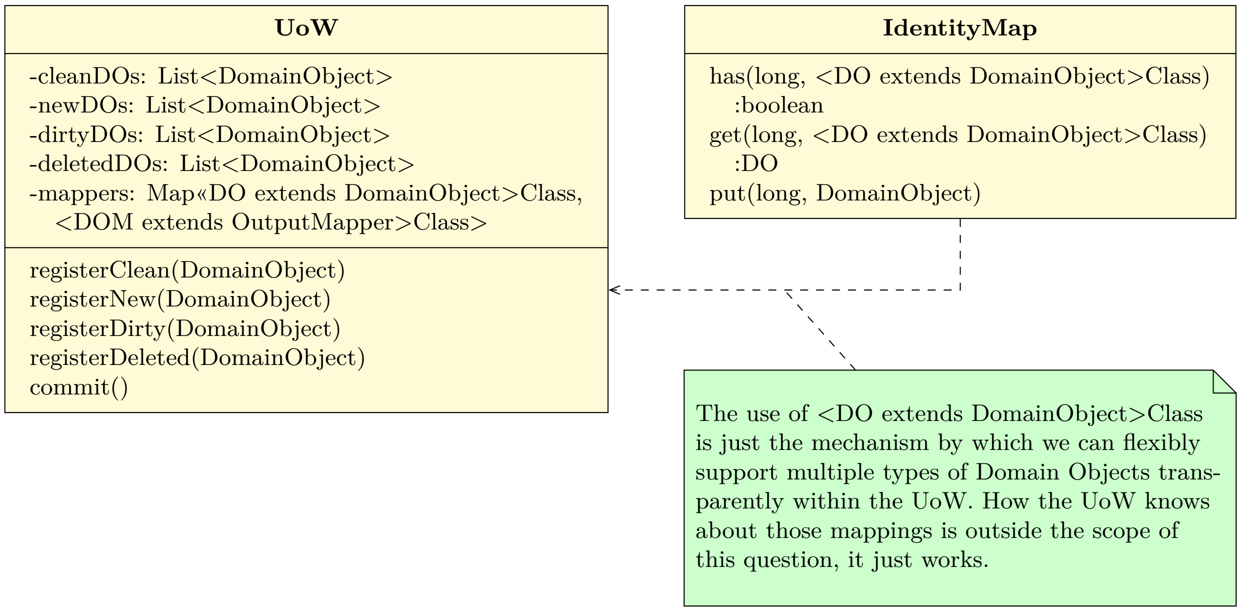
1.2.1 DomainObject, IDomainObject and DomainObjectProxy

You should use the existing support classes for Domain Objects. You should skip including fields and methods already covered in parent classes shown in a diagram unless there is some pressing need. While non-standard, I will allow the use of the stereotypes «DO», «IDO», «DOProxy» and «OutputMapper» as a shorthand to indicate relevant inheritance, which should clean up your class diagrams a bit. I would recommend using other stereotypes to identify any dominant patterns. Using these shorthands hides the use of templates/generics, but if you had a "«DO» Dog" and a "«OutputMapper» DogOM," I would know that you meant that DogOM was inheriting from OutputMapper using Dog in the template.



1.2.2 Unit of Work (UoW) and IdentityMap

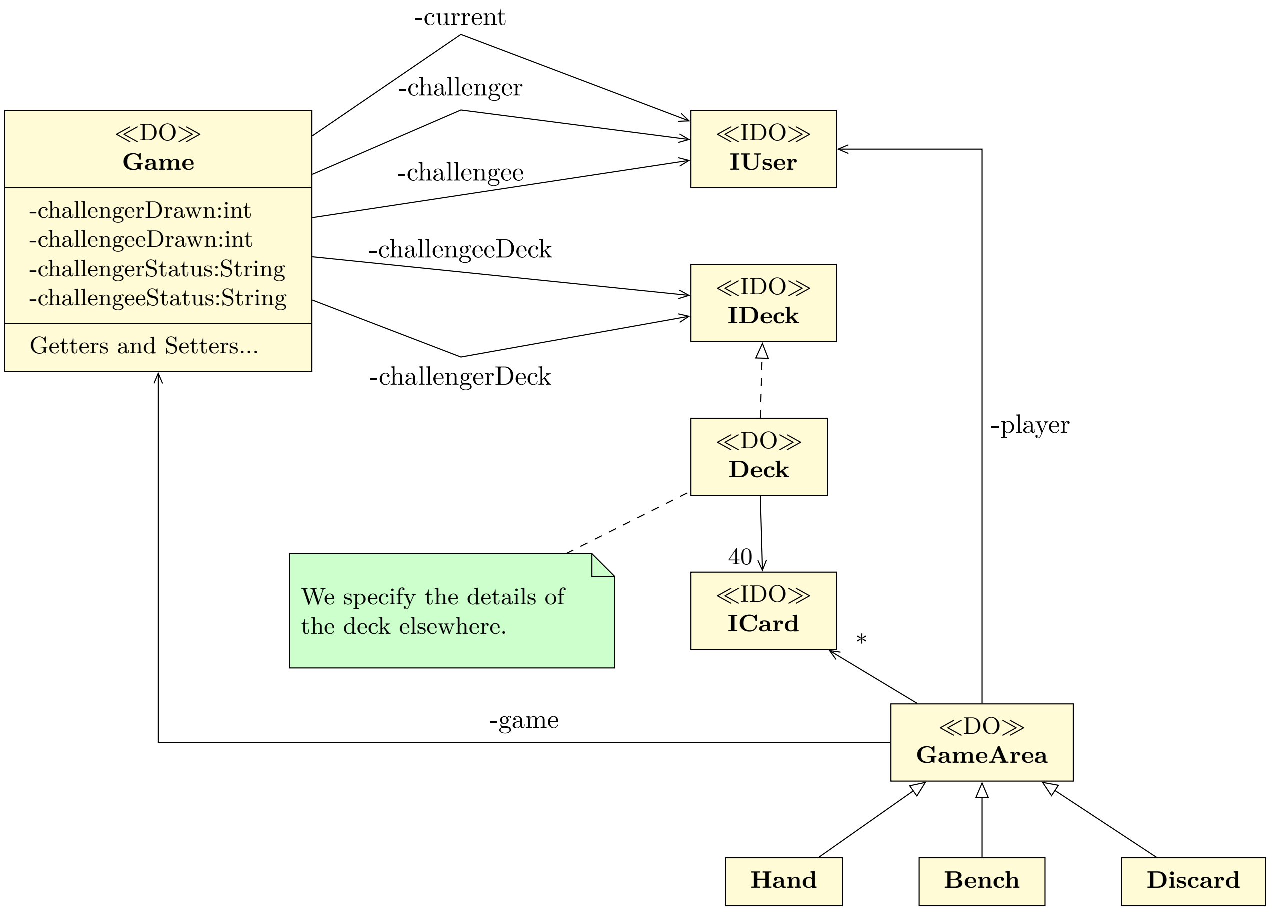
When interacting with a UoW explicitly, show a dependency on the UoW. When using the Identity Map aspect, show a dependency on the IdentityMap. Do not concern yourself with the interaction between IdentityMap and UoW.



UoW
-cleanDOs: List<DomainObject>
-newDOs: List<DomainObject>
-dirtyDOs: List<DomainObject>
-deletedDOs: List<DomainObject>
-mappers: Map«DO extends DomainObject>Class, <DOM extends OutputMapper>Class>
registerClean(DomainObject)
registerNew(DomainObject)
registerDirty(DomainObject)
registerDeleted(DomainObject)
commit()
IdentityMap
has(long, <DO extends DomainObject>Class) :boolean
get(long, <DO extends DomainObject>Class) :DO
put(long, DomainObject)
The use of <DO extends DomainObject>Class is just the mechanism by which we can flexibly support multiple types of Domain Objects transparently within the UoW. How the UoW knows about those mappings is outside the scope of this question, it just works.

1.3 Model

This model represents the existing Domain Objects in the system and how they interact. Of critical interest is "«DO» Game," which you must model in this question. Examine this diagram before proceeding.



1.4 The Full Question

Given the above model, draw a class diagram using the following patterns implemented for "«DO» Game":

- Domain Object («DO», «IDO»)
- Lazy Loading («DOProxy»)
- Domain Object Factory («DOFactory»)
- Input Mapper («InputMapper»)
- Output Mapper («OutputMapper»)
- Table Data Gateway («TDG»)
- Finder («Finder»)
- Optimistic Concurrency Management (show UML notes: where applied, and where not... like when retiring)

Make sure you also show how the following patterns incorporated into the above when considering the "View Board" Use Case:

- Front Controller («FC»)
- Dispatcher («Dispatcher» ViewBoard)
- Command («Command»)
- Template View («JSP» or «TV») (/WEB-INF/jsp/html/board.jsp)
- Unit of Work (UoW)
- Identity Map

Show all of this in three diagrams, with each diagram focusing on each of the three general layers:

- Application/Presentation
- Domain
- Data Source, Technical Services

You will be evaluated on showing the patterns, the breakdown by layer, appropriate relationships between classes/interfaces and the use of relevant and appropriate fields and the methods. You may simplify and communicate streamlining via UML notes, and you should avoid redundancy, but make sure you are complete.

2 Sequence Diagram Question

Please read this question carefully. Both the text and the diagrams contain important information needed to complete the goals of this question.

2.1 Scenario

Considering the Class Diagram you have just developed, you are now tasked with describing the behaviour of the Accept Challenge Use Case using sequence diagrams. You will be asked to create **four** sequence diagrams for the missing sections in the Global sequence diagram provided. Some code for ChallengeInputMapper is provided to give you a hint. BoardHelper's methods are provided and assumed ready to make the Template View sequence diagram more manageable (Diagram D).

Take careful note that in this implementation, ViewBoardCommand is called by AcceptChallenge after AcceptChallengeCommand completes successfully, and the JSP generates the web-page response shown later in this question.

2.2 Diagrams

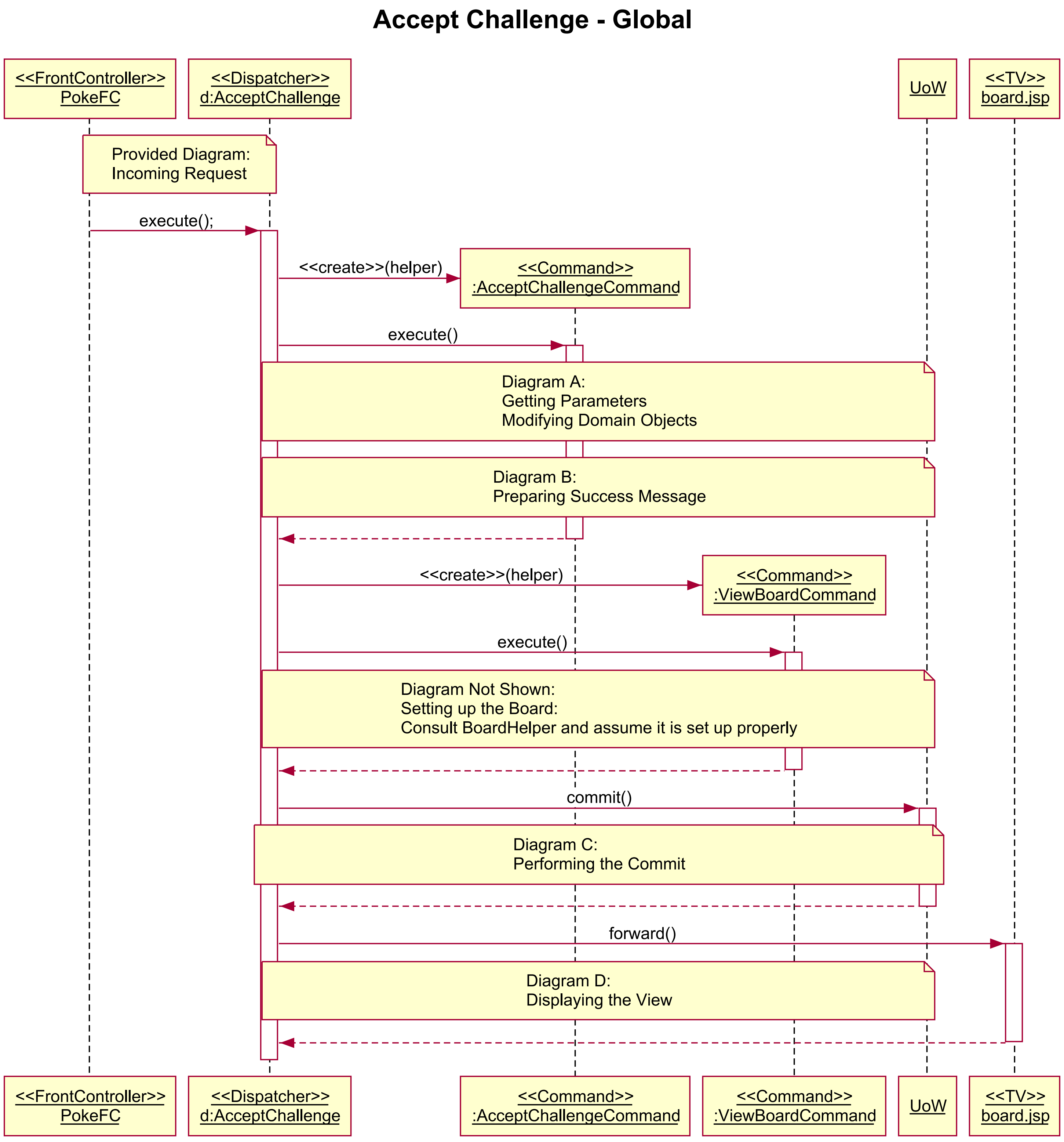

Accept Challenge - Global
<<FrontController>> PokeFC
<<Dispatcher>> d:AcceptChallenge
UoW
<<TV>> board.jsp
Provided Diagram: Incoming Request
execute();
<<create>>(helper)
<<Command>> :AcceptChallengeCommand
execute()
Diagram A: Getting Parameters Modifying Domain Objects
Diagram B: Preparing Success Message
<<create>>(helper)
<<Command>> :ViewBoardCommand
execute()
Diagram Not Shown: Setting up the Board: Consult BoardHelper and assume it is set up properly
commit()
Diagram C: Performing the Commit
forward()
Diagram D: Displaying the View

Figure 1: This Diagram should give you a global idea of how this request is being processed. Your job is to complete Diagrams A, B, C and D in a sensible way that makes use of all the appropriate patterns

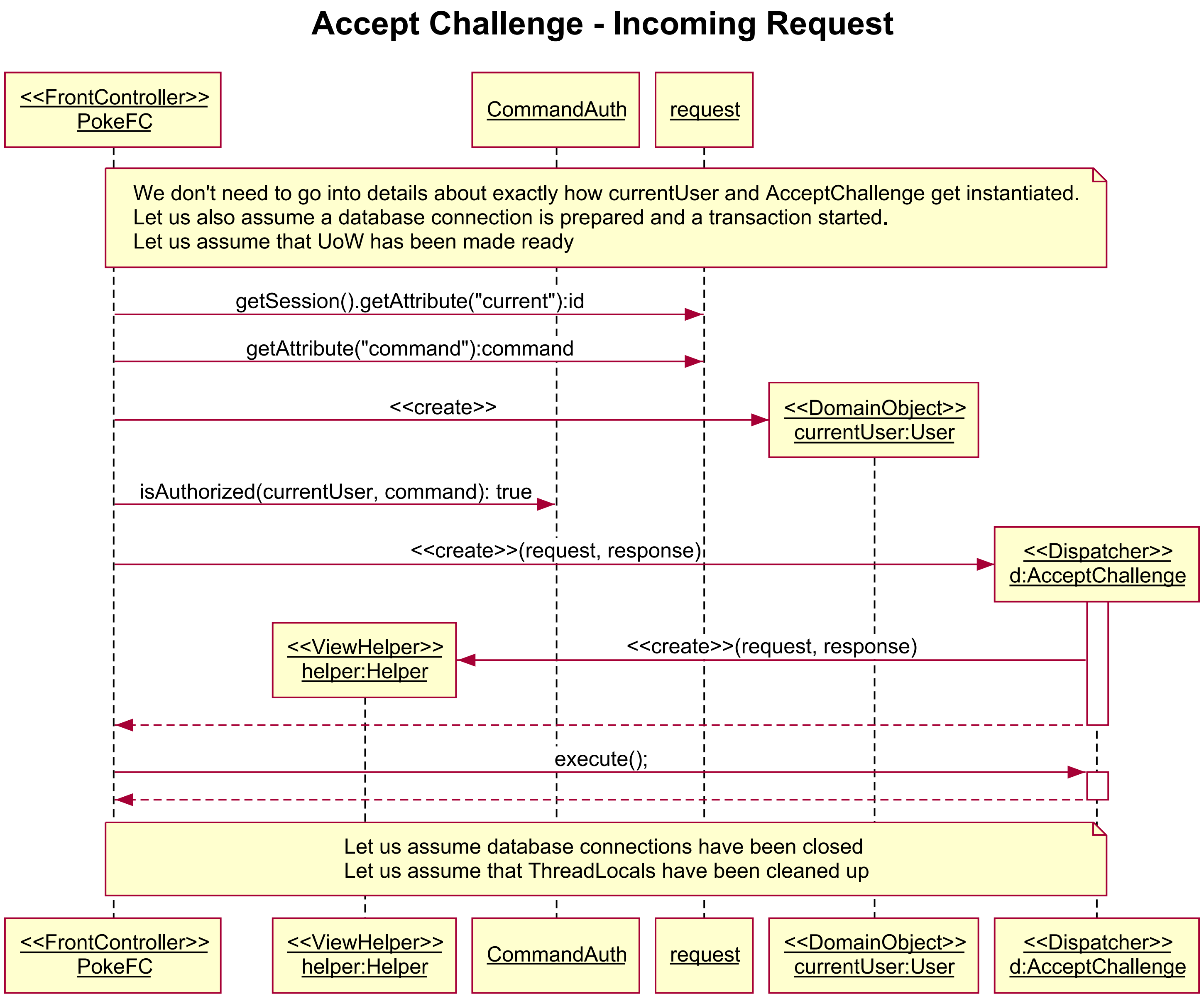

Accept Challenge - Incoming Request
<<FrontController>> PokeFC
CommandAuth
request
We don't need to go into details about exactly how currentUser and AcceptChallenge get instantiated.
Let us also assume a database connection is prepared and a transaction started.
Let us assume that UoW has been made ready
getSession().getAttribute("current"):id
getAttribute("command"):command
<<create>>
<<DomainObject>> currentUser:User
isAuthorized(currentUser, command): true
<<create>>(request, response)
<<Dispatcher>> d:AcceptChallenge
<<ViewHelper>> helper:Helper
<<create>>(request, response)
execute();
Let us assume database connections have been closed
Let us assume that ThreadLocals have been cleaned up
<<FrontController>> PokeFC
<<ViewHelper>> helper:Helper
CommandAuth
request
<<DomainObject>> currentUser:User
<<Dispatcher>> d:AcceptChallenge

Figure 2: This Diagram should give you an idea of what the Front Controller has taken care of for you. You may want to use helper and currentUser

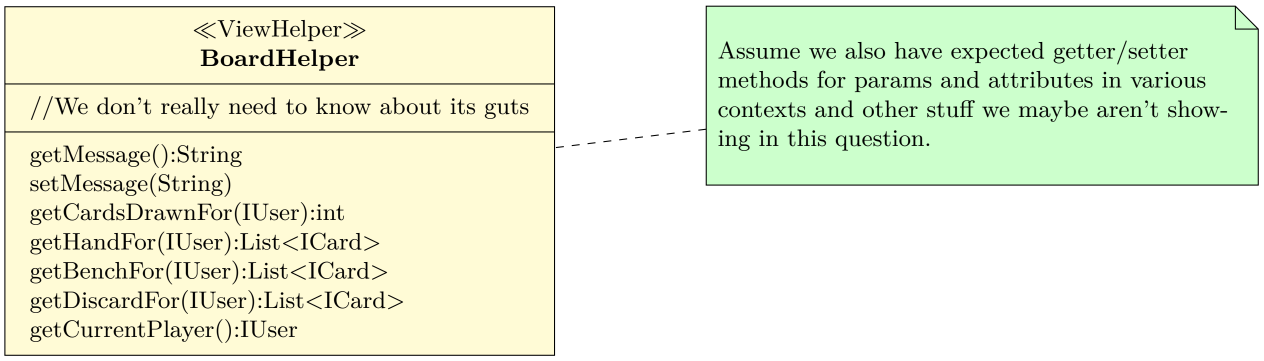

≪ViewHelper≫
BoardHelper
//We don't really need to know about its guts
getMessage():String
setMessage(String)
getCardsDrawnFor(IUser):int
getHandFor(IUser):List<ICard>
getBenchFor(IUser):List<ICard>
getDiscardFor(IUser):List<ICard>
getCurrentPlayer():IUser
Assume we also have expected getter/setter methods for params and attributes in various contexts and other stuff we maybe aren't showing in this question.

Figure 3: This is the immediately relevant part of the interface for BoardHelper.

```
public class ChallengeInputMapper implements IdentityBasedProducer {
        private static Challenge getChallenge(ResultSet rs) throws SQLException {
                return ChallengeFactory.createClean(rs.getLong("c.id"),
                                rs.getInt("c.version"),
                                new UserProxy(rs.getLong("c.challenger")),
                                new UserProxy(rs.getLong("c.challengee")),
                                ChallengeStatus.values()[rs.getInt("c.status")],
                                new DeckProxy(rs.getLong("c.deck"))
                                );
        }
```

Figure 4: This is the call used to get a Challenge from a ResultSet in ChallengeInputMapper.

```
public class AcceptChallengeCommand extends PokeCommand {
    private String getSuccessMessage(IChallenge challange)  {
        return String.format("You have accepted %s's challenge!",
            challenge.getChallenger().getUsername())
    }
```

Figure 5: This is the call used to get the success message in AcceptChallengeCommand.

2.3 Board

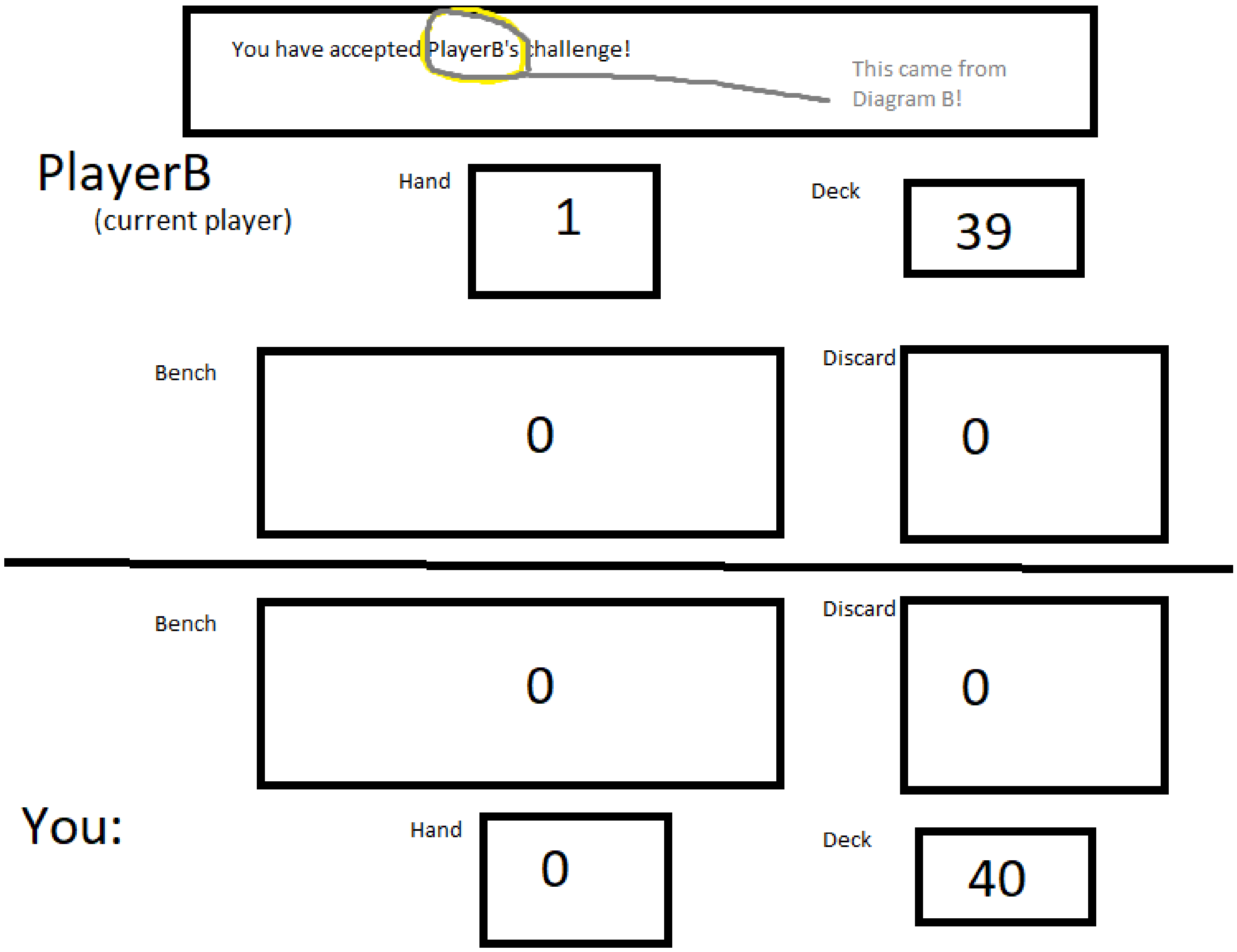


Figure 6: This is the resulting web page from the call. Note the message generated in Diagram B. In Diagram B you should be able to sensibly say how BoardHelper is being used to generate this.

2.4 The Full Question

Given the above global sequence diagram in Figure 2.2, draw a sequence diagrams to represent each of diagrams A, B, C and D.

2.4.1 Diagram A

Diagram A should show the getting of any additional parameters/attributes beyond currentUser, deck:Deck and challenge:IChallenge (which you should include as pariticpants)... mostly because I don't want to see sequence diagrams of accessing InputMappers a billion times and you'll do that in a different diagram. as well as the main domain object modification and creation. You should consider the following patterns:

- Domain Object («DO», «IDO»)
- Domain Object Factory («DOFactory»)
- Table Data Gateway («TDG»)

- Optimistic Concurrency Management
- Command («Command»)
- Unit of Work (UoW)
- Identity Map

2.4.2 Diagram B

Diagram B should show the how the success message "You have accepted PlayerB's challenge!" is generated. The code listings in Figures 2.2 and 2.2 should tell you what you're working with. You should consider the following patterns:

- Domain Object («DO», «IDO»)
- Lazy Loading («DOProxy»)
- Input Mapper («InputMapper»)
- Finder («Finder»)
- Command («Command»)
- Identity Map
- Helper («Helper»/«ViewHelper»)

2.4.3 Diagram C

Diagram C should show what happens when UoW.commit() is called. If you need to, consider the Class Diagrams Question version of UoW and OutputMapper.:

- Domain Object («DO»)
- Table Data Gateway («TDG»)
- Optimistic Concurrency Management
- OutputMapper «OutputMapper»
- Unit of Work (UoW)

2.4.4 Diagram D

Diagram D should show how the JSP makes use of the provided BoardHelper to generate the fancy web page shown in Figure 2.3. You shoould consider the following patterns:

- Domain Object («DO»)
- Template View («TV»)
- Helper («Helper»/«ViewHelper»)