Journal of King Saud University – Computer and Information Sciences 34 (2022) 10028–10043

HOSTED BY

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Software design patterns for data management features in web-based information systems

Feras Al-Hawari

Department of Computer Engineering, German Jordanian University Amman, Jordan

ARTICLE INFO

Article history:
Received 25 August 2022
Revised 22 September 2022
Accepted 5 October 2022
Available online 13 October 2022

Keywords:
Software engineering
Software design patterns
Web application
Application architecture
User interface
Data management

ABSTRACT

In complex information systems, some features may recur hundreds of times. Therefore, identifying such features and suggesting suitable design solutions for them can simplify the development and maintenance of such complex systems. In that regard, this work introduces five design patterns that were utilized to develop data management features that recurred many times in several web-based information systems used to manage enterprise and student data at the German Jordanian University. In this context, a software design pattern describes a solution to design repeating software features. The proposed design patterns are documented in a general manner using UML diagrams to enable utilizing them in different web development platforms and to allow their development using popular object-oriented programming languages. In particular, the suggested patterns seek to solve the following software features: flexible user interface for data management, reusable module for dependent dropdown filters, table data lazy loading, unified modules to handle data addition and editing, and page state restoration when navigating between related pages. Not to mention, the validation results show that the discussed design patterns were used hundreds of times while implementing six information systems for the university. Specifically, one of the patterns was utilized more than 700 times. Additionally, as it seems that some of the design patterns in this work were not investigated in related work.

1. Introduction

Enterprise and student information systems can be categorized as complex systems because they are composed of many software modules that may depend on each other. Therefore, advanced software engineering skills are needed to analyze, design, develop, and maintain such systems. Amongst those talents is the ability to identify software design patterns which specify the steps to design features that keep occurring while developing complex systems. According to Wikipedia (Wikipedia, 2022), a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Discovering and applying design patterns are very important aspects as they help in writing code faster and making the code reusable, legible, as well as maintainable.

In that regard, this paper discusses five software design patterns that were adopted while developing web-based student and enterprise information systems to manage the academic and administrative processes at the German Jordanian University (GJU). Basically, the proposed design patterns describe the design of data management related features that occurred hundreds of times while developing the following systems: Student Information System (SIS) (Al-Hawari et al., 2017; Al-Hawari et al., 2021), Human Resources System (HRS), Accounting Information System (AIS) (Al-Hawari, 2017; Al-Hawari and Habahbeh, 2020), Online Exams System (Al-Hawari et al. (2019)), Help Desk System (Al-Hawari and Barham, 2019), as well as Bus Tracking System (Al-Hawari et al., 2020). Not to mention, the proposed patterns are also applicable to data management features in web applications that are not related to university processes such as travel, shopping, and social media websites. Essentially, the advantages of the suggested design patterns in this work are as follows:

- Identifying and documenting a design pattern helps software developers in developing the related features much faster, this is because they can understand the design details better from

E-mail address: firas.alhawari@gju.edu.jo
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2022.10.003
1319-1578/

the different UML diagrams used to document the pattern and they may reuse or learn from an existing implementation of a recurring pattern.
- Implementing a design pattern sometimes results in reusable and proven classes, which can improve software reliability and reduce development costs.
- Applying a design pattern leads to more readable and maintainable code, this is because the code was developed according to a well-documented design blueprint that is clear to all developers.
- Adopting the same design for a visual feature that occurs in the information system hundreds of times leads to software modules that are consistent in behavior and layout, hence such modules will be user friendly.
- Following some design patterns leads to more flexible, efficient, and robust software modules, provided that the previous software quality attributes are amongst the expected outcomes of the adopted patterns.

Notably, this paper introduces five design patterns for different data management features in web-based information systems, as follows: the *Data Management UI Page* pattern to offer a flexible and user-friendly user interface for data management, the *Dependent Dropdown Filters* pattern to introduce reusable modules for cascaded dropdown filters, the *Table Data Lazy Loading* pattern to simplify the development of table data lazy loading, the *Page State Restoration* pattern to support page state restoration when navigating between interdependent pages, as well as the *Add and Edit Page* pattern to handle both data addition and editing via unified modules. Reportedly, the *Dependent Dropdown Filters* as well as the *Add and Edit Page* patterns were not discussed in the related work, while the other three patterns were not applied in the context of data management in web applications elsewhere.

The remainder of the paper is structured as follows. In Section 2, a literature review discussing existing work in the field of software design patterns is conducted. In Section 3, the identified five software design patterns targeting the design of data management pages in web-based applications are discussed. In Section 4, the design patterns are validated in six enterprise web-based applications, besides the comparison results of this work to related studies are illustrated. In Section 5, the conclusions of this study are pointed out.

2. Literature review

Software design patterns have been addressed by several authors over the past years. For instance, the popular book by (Cooper, 2000) discussed 23 object oriented design patterns that fall under the following three categories: creational, structural, and behavioral. The book by (Crawford and Kaplan, 2003) covered real world J2EE design patterns. The book by (Fowler, 2012) discussed 10 design patterns suitable for large enterprise applications. The study in (Heer and Agrawala, 2006) covered 12 design patterns for information visualization applications. The book by (Joshi, 2016) presented various design patterns categories for ASP.NET developers. In the work in (Burns and Oppenheimer, 2016), three types of design patterns observed in container-based distributed systems were demonstrated. In the study in (Fernandes and Jose, 2017), several architectural patterns for Android application development were analyzed. The patterns for enterprise application integration were evaluated in the survey in Ritter et al. (2017). The study in (Haq et al., 2017) introduced a design pattern for secure information systems. The book by Ayeva and Kasampalis (2018) explained several design patterns using real world examples coded in Python. In the studies in Tkaczyk et al. (2018) and Bloom et al. (2018), design patterns for Internet of Things (IoT) and Industrial IoT were cataloged, respectively. A design pattern to extract a statistical report was introduced in Mythily et al. (2019). Automated approaches for classifying and detecting software design patterns were also proposed in Hussain et al. (2019) and Singh et al. (2021). The study in Malik et al. (2020) introduced UI design patterns for flight reservation websites. The work in Guest et al. (2019) covered design patterns for augmented reality systems. A survey on the adoption of software design patterns for the cloud was conducted in Sousa et al. (2021). The work in Lu et al. (2021) and Rajasekar et al. (2020) discussed design patterns for blockchain based applications. The study in Washizaki et al. (2022) identified 15 design patterns suitable for machine learning applications, besides the studies in Lu et al. (2022) and Martínez-Fernández et al. (2022) explored some design patterns for artificial intelligence (AI) systems. The work in Bach et al. (2022) presented design patterns for dashboards.

Based on the aforementioned review and as asserted in the study in Mayvan et al. (2017), it is obvious that most researchers focused on topics related to pattern recognition, pattern development, and pattern application. Specifically, the introduced design patterns targeted object oriented software design in the early years, whereas they recently started focusing on attractive areas such as blockchain, AI, machine learning, augmented reality, and IoT. Also, the topic of using different programming languages such as Java, C#, C++, and Python to develop design patterns was covered. Moreover, the utilization of design patterns in different web application development platforms such as J2EE and ASP.net was addressed. Additionally, the subject of applying design patterns in different environments such as distributed systems and the cloud was discussed. In that regard, this work introduces five important software design patterns to aid in designing features that kept recurring while developing data management pages in web-based applications. The patterns are described in a general manner using UML diagrams to allow applying them in various web development platforms and to enable their implementation using different programming languages. To the best of our knowledge the proposed design patterns in this work were either not discussed or not tied to data management pages by others as will be later shown in subsection 4.2.

3. Data management design patterns

The five design patterns for the features that kept repeating while developing data management pages in Java Enterprise Edition (Java EE) (Juneau, 2013) web applications are discussed in the following subsections. Noting that such patterns can be also utilized to develop web-based systems in other similar web development frameworks and can be implemented in several object oriented programming languages. In that context, a Java EE web application is designed according to a three-tier architecture that consists of three tiers (i.e., layers) namely the client tier, web tier, and data tier as shown in Fig. 1. The client tier is the layer where the user can access the application and then navigate through its pages using a web browser running on the client's device. The web tier is the layer at which the application components are managed by the web container of the Java EE application server running on a dedicated host. The data tier is the layer where the application information is managed in databases through a database management system (DBMS) hosted on a separate server. Not to mention, the web application running in the web tier consists of the following components: JavaServer Faces (JSF) (JSF, 2022) pages, managed beans, and Data Access Objects (DAOs). A JSF page represents a UI screen in the web application by including HTML, JSF, and Primefaces (PrimeTek, 2022) UI elements such as text fields, checkboxes,

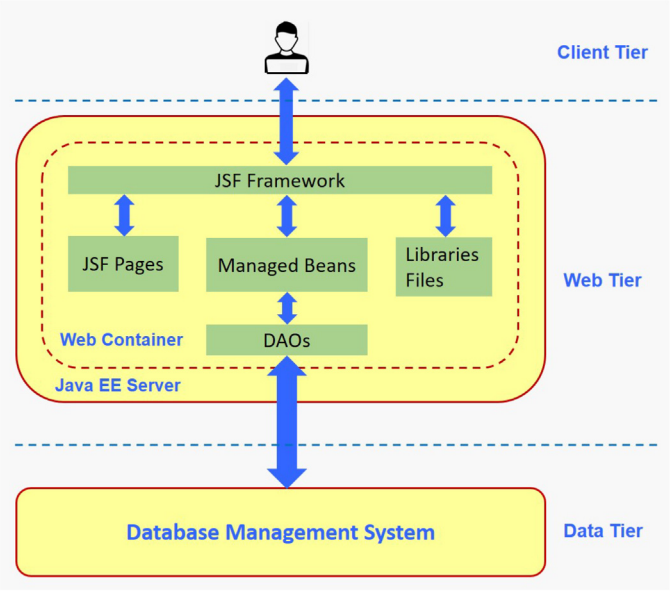

Fig. 1. A three tier web application architecture.

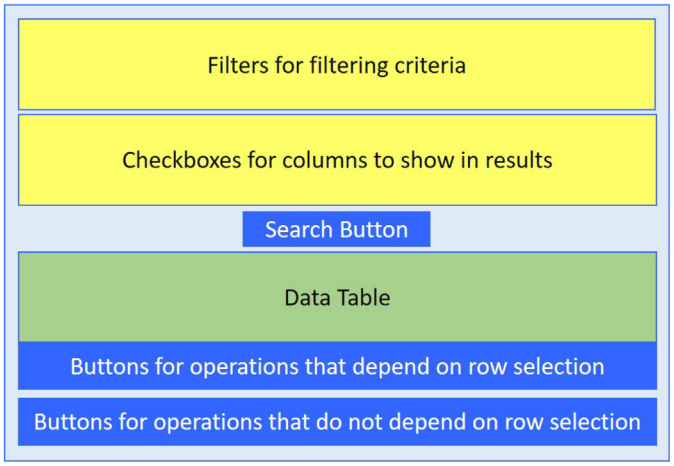

Fig. 2. A suggested page layout for a data management web page.

dropdown menus, data tables, calendars, field sets, and panels. A managed bean is a special Java class that is managed (i.e., instantiated, serviced, and destroyed) by the JSF framework. A managed bean instance can be referenced in a JSF page to save page state as well as support page logic. It further utilizes DAOs to interact with the databases in the data tier via the Java Database Connectivity (JDBC) API methods.

3.1. Data management UI page pattern

It was noticed in the developed web applications that the UI components needed in pages used for data management are almost the same regardless of the managed data types. Therefore, it is very important to identify those components and to propose how they are arranged on a page to make sure that all data management pages are harmonious in behavior and layout. Accordingly, users can utilize such pages easily because they can anticipate the locations and actions of the repeated UI components. Further, system developers would save a lot of time by following the proposed design pattern when implementing any new page for data management purposes.

In that context, a flexible page layout for a data management screen is suggested in Fig. 2, whereas an example for a data management page that is designed accordingly is illustrated in Fig. 3. Based on that, the page is comprised of a field set element containing filtering criteria components, a field set element with checkboxes to control which table columns to display, a search button to update the data table with the rows that match the chosen filtering criteria, a data table component with a footer containing buttons to manipulate the table data, and a panel element to place buttons for the operations that do not depend on the table data.

The filtering criteria field set element (see Fig. 3) may contain different types of UI components, as necessary, to allow retrieving the desired data from the database. It may include dropdown menus, text fields, checkboxes, and calendar components as needed to specify the values of the table columns to be used in finding the table rows that match the search criteria. Whilst the next field set element, that is named "Columns to Show in Results" as shown in Fig. 3, contains checkboxes to control which table columns will be displayed/hidden upon every search operation. The ability to hide some of the table columns is helpful in case the table has many columns while the user needs to only view a few of them to analyze the data.

The data table is a commonly used UI component (see Fig. 3) in web applications to view and manipulate a list of entities. Each entity in the list is represented by a row in the table and an entity attribute may relate to a column in the table. In most UI frameworks, the data table component provides vertical/horizontal scroll bars when the loaded content is long/wide to display in the screen. Also, it supports a pagination feature to split large data into multiple pages and to facilitate easy navigation between those pages. Further, it enables sorting the rows according to a specific column and permits searching the loaded rows based on the filters values associated with the columns. Besides, it offers an option to operate on a single row or multiple rows.

Not to mention, managing the list of entities in a data table demands providing operations in the screen to edit, delete, and view a selected entity (i.e., row) when necessary. Further, an operation to add an entity to the list is also needed in many applications. Besides, other actions could be introduced in the screen to meet desired system functionality. In that context, it is worth noting that the data manipulation operations (i.e., edit, delete, and add) ordinarily result in altering the information in the data table as well as the database. In addition, performing some operations (e.g., edit, view, or delete) requires row selection first, therefore it is appropriate to place such buttons on the table footer to highlight that dependency. Conversely, any button for an operation that does not depend on row selection (e.g., the *Add* button) should be located outside the table.

3.2. Dependent dropdown filters pattern

The filtering criteria field set utilized in several pages in a web application could contain a repeated group of filters that allow users to search for the same types of data in those pages. Particularly, a group of dependent dropdown filters is a common pattern of recurring filters. In this scenario, the values to populate a child dropdown filter depend on the selected item in the parent dropdown filter. For example, the entity relationship (ER) diagram in Fig. 4 illustrates that a faculty is comprised of multiple departments, whereas a department could offer several majors. Based on that, the three dependent dropdown filters shown in Fig. 5 would be needed to search for faculties, departments, and majors in related data management pages. Whilst the departments list would be populated upon selecting a faculty from the faculties list, whereas the majors list would be built after selecting a department

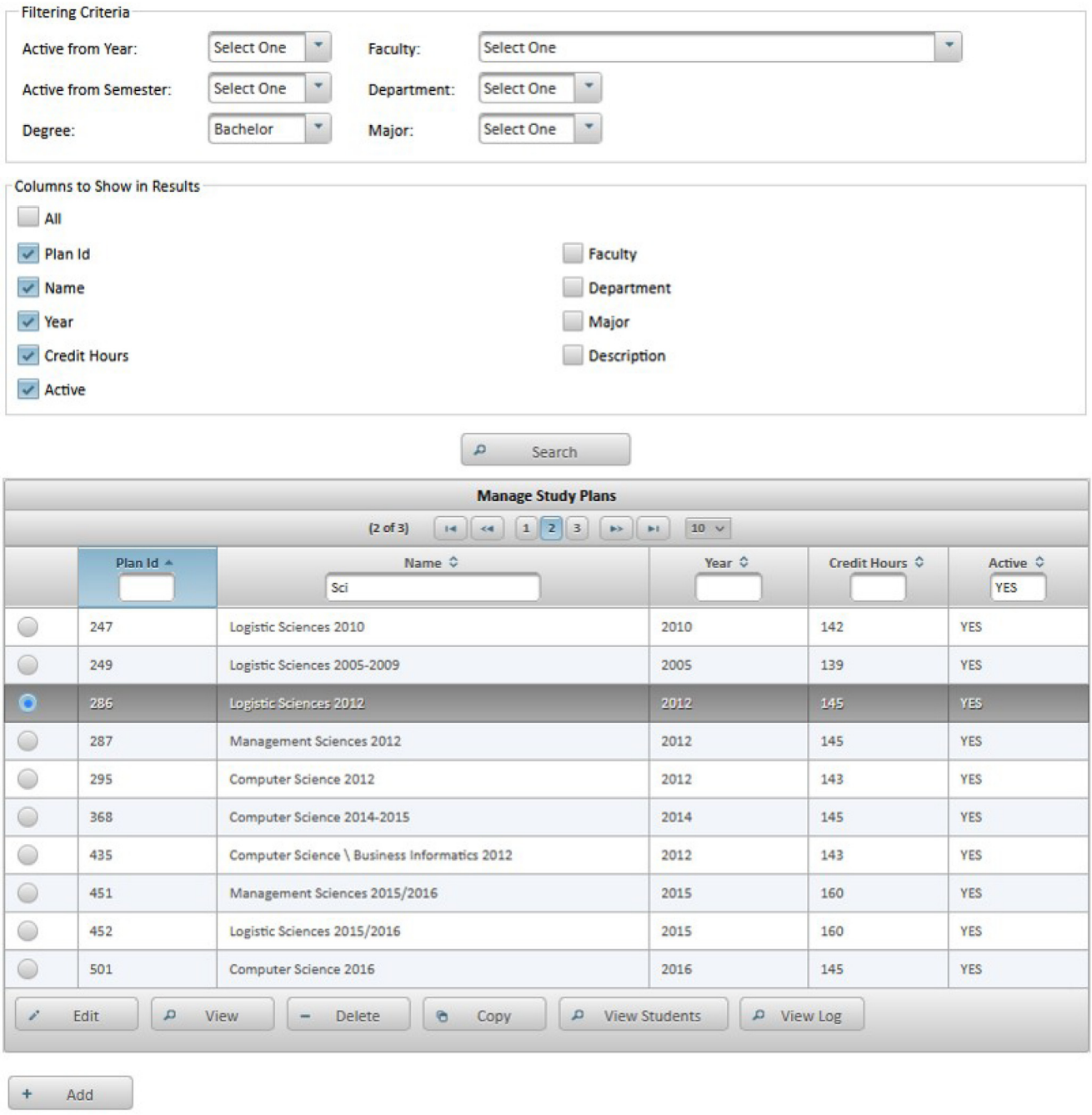

Fig. 3. An example on a data management page where the UI elements are arranged on it according to the page layout in Fig. 2.

from the departments list. Not to mention, the dependent dropdown filters can be used to model other related lists such as years versus semesters, sponsors versus scholarships, as well as countries versus cities.

The use of the same dependent dropdown filters in different pages requires repeating the code to manage those filters in the managed bean classes of the related pages. At the same time, repeating code in different classes is time consuming and could cause maintenance problems. Therefore, to overcome such issues, it is suggested to introduce a utility class for each group of dependent dropdown filters to be reused in all dependent classes when needed. Suitably, the class diagram in Fig. 6 shows the basic classes in this design pattern. Accordingly, the *DependentFilters* class contains methods (e.g., *buildLevel1Items* to *buildLevel4Items*) to build lists of items to populate the corresponding filters at each level. Hence, the number of such methods is equal to the number of dependent filters (i.e., four filters in this case) in the *DependentFilters* class, whilst each method utilizes a related method (e.g., *getLevel1Items* to *getLevel4Items*, respectively) in the *DependentFiltersDao* class to retrieve its items data from the database. Consequently, the *DependentFilters* class can be reused in the managed beans (e.g., *Page1Bean* and *Page2Bean*) associated with all pages that utilize the corresponding dependent filters. Not to mention, the method to initialize the values of the first-level filter must be called when the *init* method of its related managed bean is invoked. Whereas a method to build the list for any of the other filters should be triggered upon selecting an item in its parent filter list.

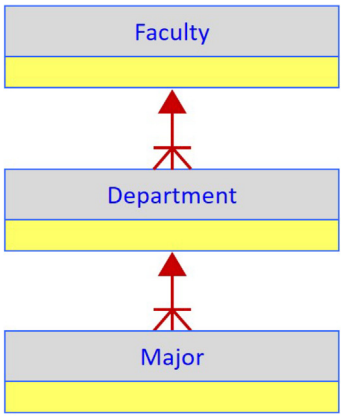

Fig. 4. An ER diagram for three related database tables to manage faculties, departments, and majors.

Specifically, a class named *FacultyFilters* can be developed to represent and manage the three dependent dropdown filters shown in Fig. 5. The class diagram representation for the *FacultyFilters* class is shown in Fig. 7, where three lists (e.g., *faculties*) are present to store the filters values, three attributes (e.g., *selectedDepartmentId*) are utilized to capture the identifiers of each selected item in the filters, a reference to a DAO needed to access the database is saved in the *facultyFiltersDao* attribute, and three methods (e.g., *buildMajors*) are available to build the filters lists.

The sequence diagram in Fig. 8 illustrates objects interactions pertaining to updating three faculty related dropdown filters within a web page to manage an associated entity type. Accordingly, when such a page is requested, the JSF framework instantiates the managed bean (e.g., *manageEntityXBean*) of the page, subsequently it invokes the *init* method on that managed bean. In turn, the *init* method instantiates the *facultyFilters* object that references a *facultyFiltersDao* object, then it invokes the *buildFaculties* method on it to generate the faculties list. In case a faculty was selected before, the *buildDepartments* method will be also invoked on the *facultyFilters* object to populate the departments list. Whilst, when a new faculty or department is selected, the browser sends an AJAX (Asynchronous JavaScript And XML) request to the JSF framework to efficiently update the affected dropdown filters values without the need to reload the whole page. For example, when a new faculty is selected, the *buildDepartments* method will be invoked on the *facultyFilters* object to populate the departments list. Furthermore, the *buildMajors* method will be called to rebuild the majors list upon choosing a different department. It is worth mentioning that the *buildFaculties*, *buildDepartments*, and *buildMajors* methods call the corresponding methods (e.g., *getFaculties*) of the *facultyFiltersDao* object to obtain the required data, correspondingly the DAO methods utilize the needed JDBC API methods to communicate with the database to build the related lists.

3.3. Table data lazy loading pattern

The table data lazy loading design pattern is commonly used in web application development to defer the loading of the data table

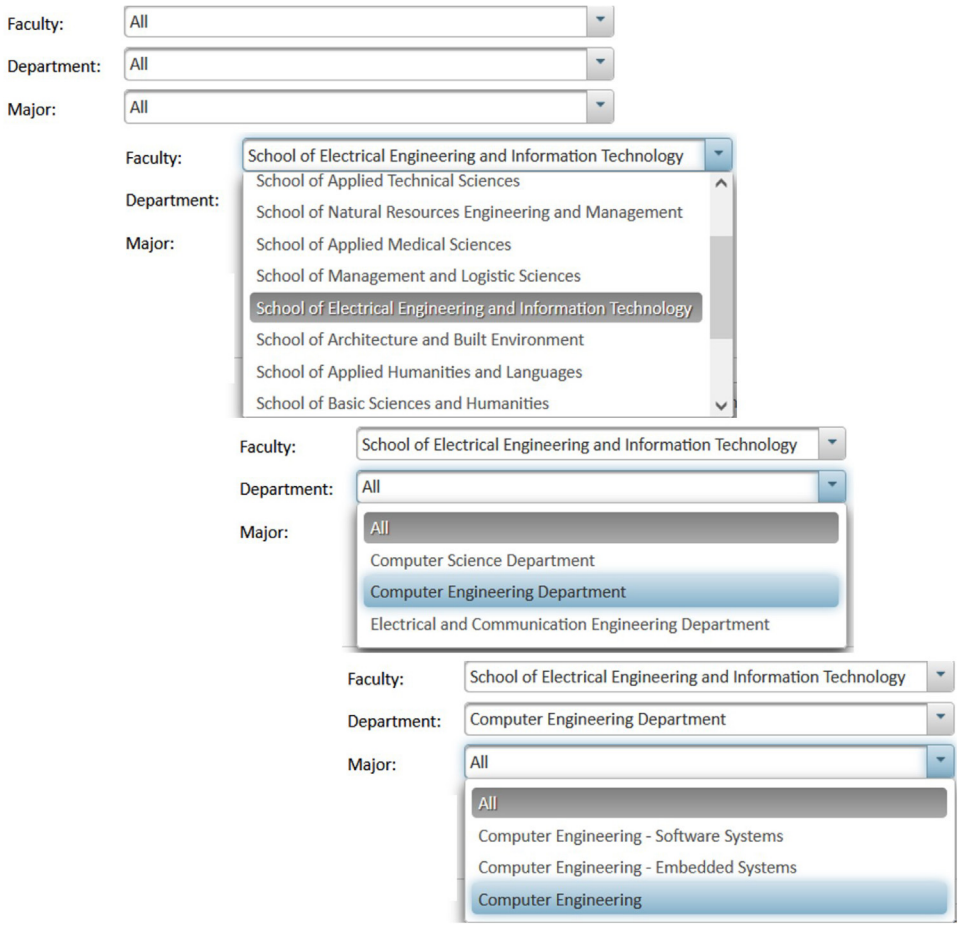

Fig. 5. An example on three dependent dropdown filters to search for faculties, departments, and majors in a web page.

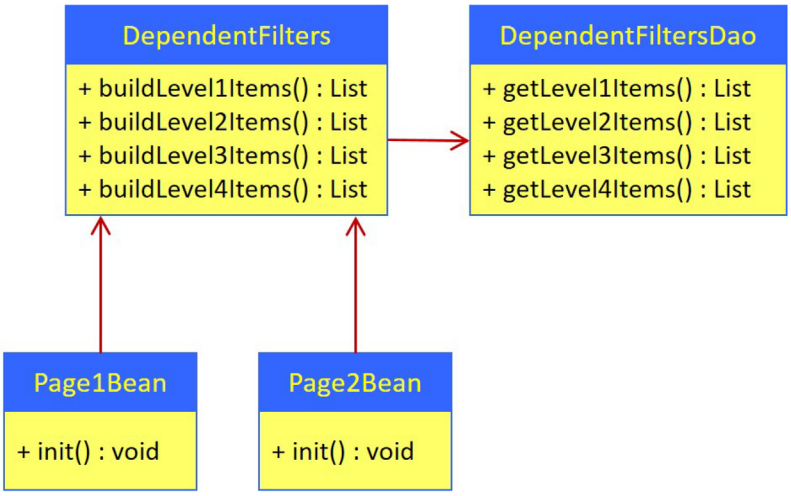

Fig. 6. A class diagram for the classes needed to implement four dependent dropdown filters used in two web pages.

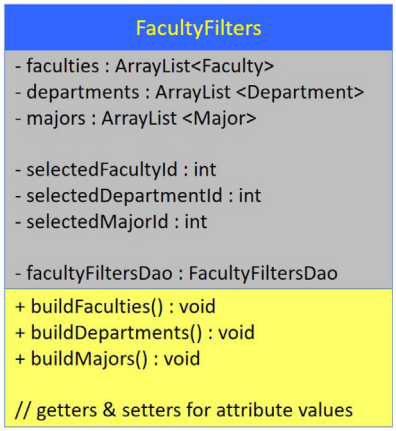

Fig. 7. A class diagram for the utility class to manage the dependent faculty dropdown filters.

rows until they are requested. Accordingly, only the rows in the active page (e.g., page 2 in Fig. 3) in the data table are loaded and displayed. Whilst the rest of the data table rows are loaded on demand, that is when their respective page is selected. Based on that, the main advantage of this design pattern is reducing the load time of a web page that contains a data table with thousands of rows.

Various web application frameworks like PrimeFaces (PrimeTek, 2022) support table data lazy loading. However, using the built-in packages to support lazy loading in a web application page is not straight forward as it requires developing several classes and methods to implement a data table with lazy loading capabilities. Therefore, the importance to introduce documented templates and utilities for that purpose in the development environment has been addressed in this work to aid programmers in implementing such a feature in a simple and quick manner.

The class diagram in Fig. 9 illustrates the needed classes to implement table data lazy loading to manage two entity types (i.e., *EntityA* and *EntityB*) in different web pages. Accordingly, each table web page is associated with a managed bean that has a variable of type *LazyDataModel* to store the table data. Specifically, a managed bean of type *ManageEntityABean* is associated with the web page containing a table to manage objects of type *EntityA*. Whereas, the *LazyEntityADataModel* and *EntityADao* classes must also be developed to support lazy loading of objects of type *EntityA* within the data table. The *LazyEntityADataModel* class should extend the *LazyDataModel* class from the PrimeFaces *model* package, and hence it must override the *load* method in the parent *LazyDataModel* class. The *EntityADao* class implements two methods to be used by the *load* method in the *LazyEntityADataModel* class to access the information for entities of type *EntityA* from the database. Particularly, it provides a method named *buildLazyList* to retrieve the data of each visible row in the active table page. It also offers another method named *getListCount* to obtain the table total rows count. Additionally, the *LazyDataModelSqlsUtil* class is introduced to relieve programmers from the burden of developing the required code to generate complex SQL statements such as those in Fig. 10 and Fig. 11 that the different DAOs need to access the respective entities data from the database.

The sequence diagram in Fig. 12 illustrates objects interactions to display a web page that contains a data table with lazy loading capabilities. Accordingly, when the web page is requested, the JSF framework instantiates the managed bean named *manageEntityABean* that is associated with the page, then it invokes the *init* method on the managed bean. In turn, the *init* method initializes the *data* variable in the managed bean to an instance of the *LazyEntityADataModel* class. The *lazyEntityADataModel* object gets a handle to an *entityADao* object, while the *entityADao* object contains a reference to the *lazyDataModelSqlsUtil* object. When the managed bean is initialized, the JSF framework invokes the *load* method on the *lazyEntityADataModel* object to build a list of the *EntityA* objects (i.e., rows) to be displayed in the active table page. The *load* method invokes the *buildLazyList* method and then the *getListCount* method on the *entityADao* object to initialize the *data* list in the *lazyEntityADataModel* object and the *rowCount* value (via the *setRowCount* method) in the parent *lazyDataModel* object, respectively. Whereas, the *buildLazyList* and *getListCount* methods first utilize the methods of the *lazyDataModelSqlsUtil* object to obtain the SQL queries to select the needed data from the database, and they subsequently use the JDBC API methods to execute the generated SQL queries on the database to build the needed results. Ultimately, the JSF framework produces an HTML representation for the table web page, then it dispatches that within a HTTP response message to the browser for display.

3.4. Page state restoration pattern

To reduce memory usage on an application host, it is recommended to configure most managed beans in a Java EE application

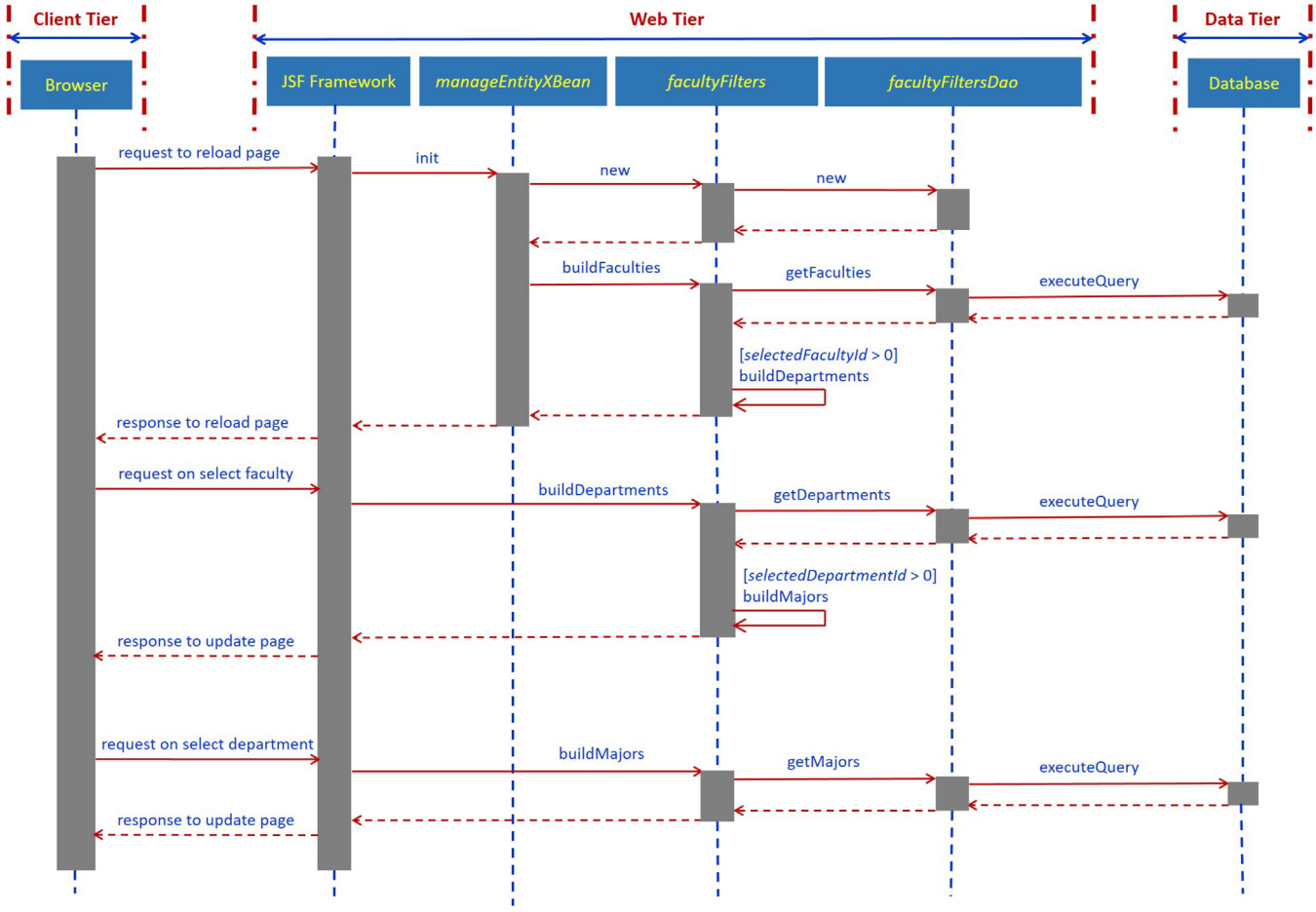

Fig. 8. A sequence diagram that illustrates objects interactions to update three faculty related dependent dropdown filters in a web page.

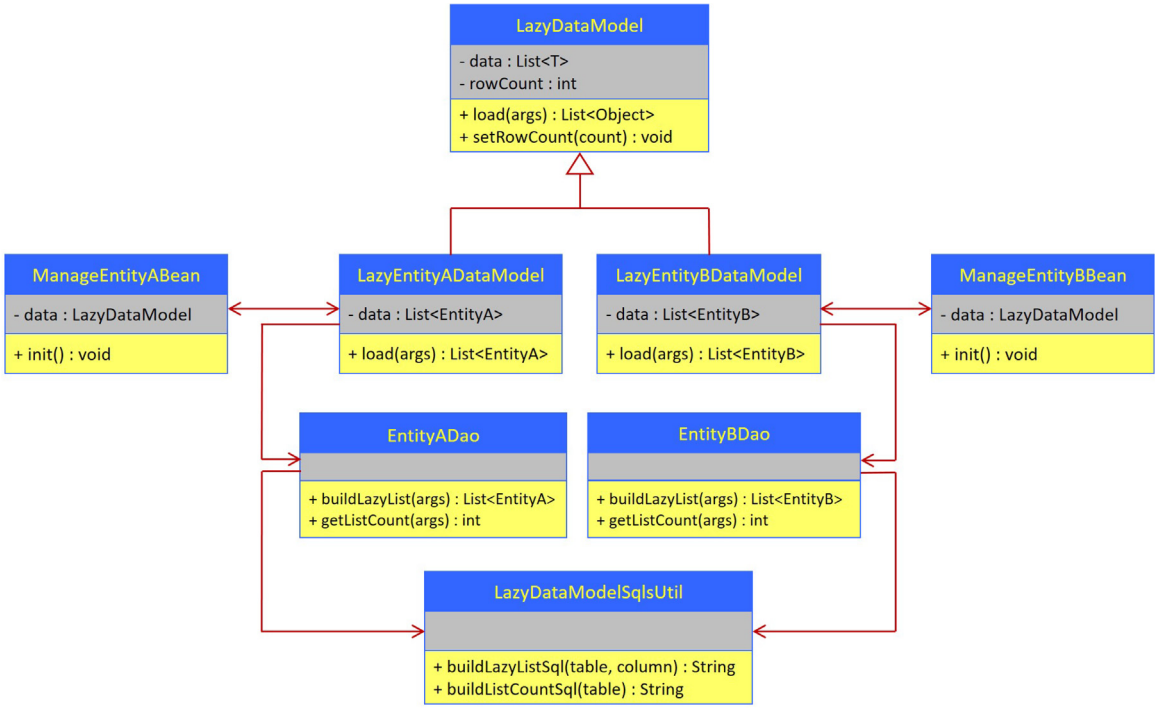

Fig. 9. A class diagram for the classes needed to implement table data lazy loading to manage two entities in different web pages.

```
1.  SELECT *
2.  FROM
3.    (SELECT data.*, ROWNUM RNUM
4.     FROM
5.      (SELECT *
6.       FROM
7.        (SELECT * FROM STUDY_PLAN
8.         -- Sort by the PLAN_ID column in descending order
9.         ORDER BY PLAN_ID DESC
10.       )
11.      -- Filter by the ACTIVE and NAME_EN filter values
12.      WHERE UPPER(ACTIVE) LIKE UPPER('%YES%')
13.        AND UPPER(NAME_EN) LIKE UPPER('%Sci%')
14.     ) data
15.    WHERE ROWNUM <= 20      -- <= offset + pageSize
16.   )
17. WHERE RNUM > 10;          -- > offset
```

Fig. 10. The SQL statement produced by the *buildLazyListSql* method to access the data for the study plans in page 2 of the data table shown in Fig. 3; correspondingly the data is retrieved from the STUDY_PLAN database table when the Name filter value is "Sci", the Active filter value is "YES", the rows are sorted by Plan Id in descending order, and the row number is between 11 and 20 (assuming a page has 10 rows in this example) as specified in Fig. 3. Assuming in this case that no filtering criteria is selected and all table columns are to be shown.

```
1.  SELECT COUNT(*)
2.  FROM
3.    (SELECT *
4.     FROM
5.      (SELECT * FROM STUDY_PLAN)
6.     -- Filter by the ACTIVE and NAME_EN filter values
7.     WHERE UPPER(ACTIVE) LIKE UPPER('%YES%')
8.       AND UPPER(NAME_EN) LIKE UPPER('%Sci%')
9.    );
```

Fig. 11. The SQL statement produced by the *buildListCountSql* method to obtain the count of the study plans in the data table shown in Fig. 3 when the Name filter value is "Sci" and the Active filter value is "YES" as entered in Fig. 3. Assuming in this case that no filtering criteria is selected and all table columns are to be shown.

as view scoped (using the *@ViewScoped* annotation), whereas few managed beans could be defined as session scoped (via the *@SessionScoped* annotation). This is justifiable because the JSF framework promptly destroys an instance of a *ViewScoped* bean when the user navigates away from its related page, while it keeps an instance of a *SessionScoped* bean as long as its HTTP session is valid. Accordingly, a *ViewScoped* bean is utilized to retain state (e.g., data table selections and filtering criteria choices) throughout the lifetime of the page that referenced it, whilst a *SessionScoped* bean is useful to store data (e.g., username and user preferences) that is shared amongst several pages. Therefore, the navigation from a data management page (e.g., the page in Fig. 3) to any of its related pages (e.g., Add page) will result in discarding all the user selections (e.g., selected data table row, picked data table page, and filtering criteria choices) in case the state of the source page is only stored in a *ViewScoped* bean instance. Consequently, when the user navigates from the related page to the source data management page, then all previous user selections in the source page will be reset to their default values. This is considered user unfriendly especially when the number of selections to be redone in the source page is large. So, to enhance application usability and facilitate code reuse, it is very important to adopt a design pattern for restoring the state of a data management page when it is reloaded due to navigating back to it from a related page.

An instance of a *SessionScoped* bean named *PagesStateBean* and shown in Fig. 13 can be utilized in every session to manage the information needed to support page state restoration. Besides, a new class should be developed for every different data management page to allow saving the state of the filtering criteria as well as the checkboxes that control which table columns to display. Not to mention, the data table state can be separately captured in a reusable class named *DataTableState* because the attributes to save the table state are the same for any data management page. Based on that, every new page state class should extend (i.e., reuse) the *DataTableState* class to store the state of the data table. Then, a reference to every new page state class should be also added in the *PagesStateBean* class to allow accessing the page state across different screens.

For example, the *ManageStudyPlansPageState* class (see Fig. 13) that extends the *DataTableState* class should be introduced and later referenced in the *PagesStateBean* to allow restoring the state of the page shown in Fig. 3 when needed. An instance of the *ManageStudyPlansPageState* class is utilized to store the identifiers (e.g., *selectedYearId*, *selectedDegreeId*, and *selectedMajorId*) for the selected items in the dropdown filters inside the filtering criteria field set, besides it is used to save the boolean values (e.g., *showPlanId*, *showActive*, and *showFaculty*) for the checkboxes within the "Columns to Show in Results" field set. Whilst the data table state is saved in the parent *DataTableState* instance as follows: the index of the first row in the selected page is saved in the *offset* attribute, the number of rows per page is stored in the *pageSize* attribute, the name of the sorted column is captured in the *sortColumn* attribute, the sort order whether it is ascending or descending is specified in the *sortOrder* attribute, and the values entered in the table filters fields are saved in the *filters* hash map. Not to mention, the *clear* methods in the aforementioned two classes can be used to reset the related attributes values to their defaults. Further, the *getPageState*, *setPageState*, and *clearPageState* methods in the *PagesStateBean* class can be used to get a page state class instance, store a page state instance, and clear the attributes values of a page state instance, respectively. The previous three methods can determine which instance to manage based on the value of the passed *pageName* parameter, given that each new page state class should have a predefined name added in the ENUM class named *PageNameEnum*.

The class diagram shown in Fig. 14 demonstrates all the classes needed to develop this design pattern. In general, the *PagesStateBean* class is utilized to save the state of all source pages for data management. Each source page is associated with a managed bean named *SourcePageBean* that references a class named *SourcePageState* which is used to capture the page state. The *SourcePageState* class extends the *DataTableState* class as previously explained. Whilst the *navigateToRelatedPage* method in the *SourcePageBean* class is required to navigate from the source page to its related page. Before handling the navigation, that method has to save the source page state that is captured in the *SourcePageState* instance in the *PagesStateBean* instance using the *setPageState* method. The related page is associated with a *RelatedPageBean* class that implements a *navigateToSourcePage* method to allow navigating back to the source page. That method should set the *restoreState* flag in the *SourcePageState* instance to true to indicate that the state of the source page has to be restored when revisited. In turn, the *init* method in the *SourcePageBean* class decides whether to restore the source page state based on the value of the *restoreState* flag.

The sequence diagram in Fig. 15 illustrates objects interactions to load a source page, navigate from a source page to a related page, and go back from the related page to the source page. Firstly, when the source page is initially requested, the JSF framework invokes the *init* method on its managed bean instance (e.g., *sourcePageBean*). In turn, the *init* method invokes the *getPageState* method on the *pagesStateBean* instance to obtain the stored instance of the source page state class. The handle to the obtained instance is saved in a variable named *pageState*. Then, the *isRestoreState* method defined in the *DataTableState* class is invoked

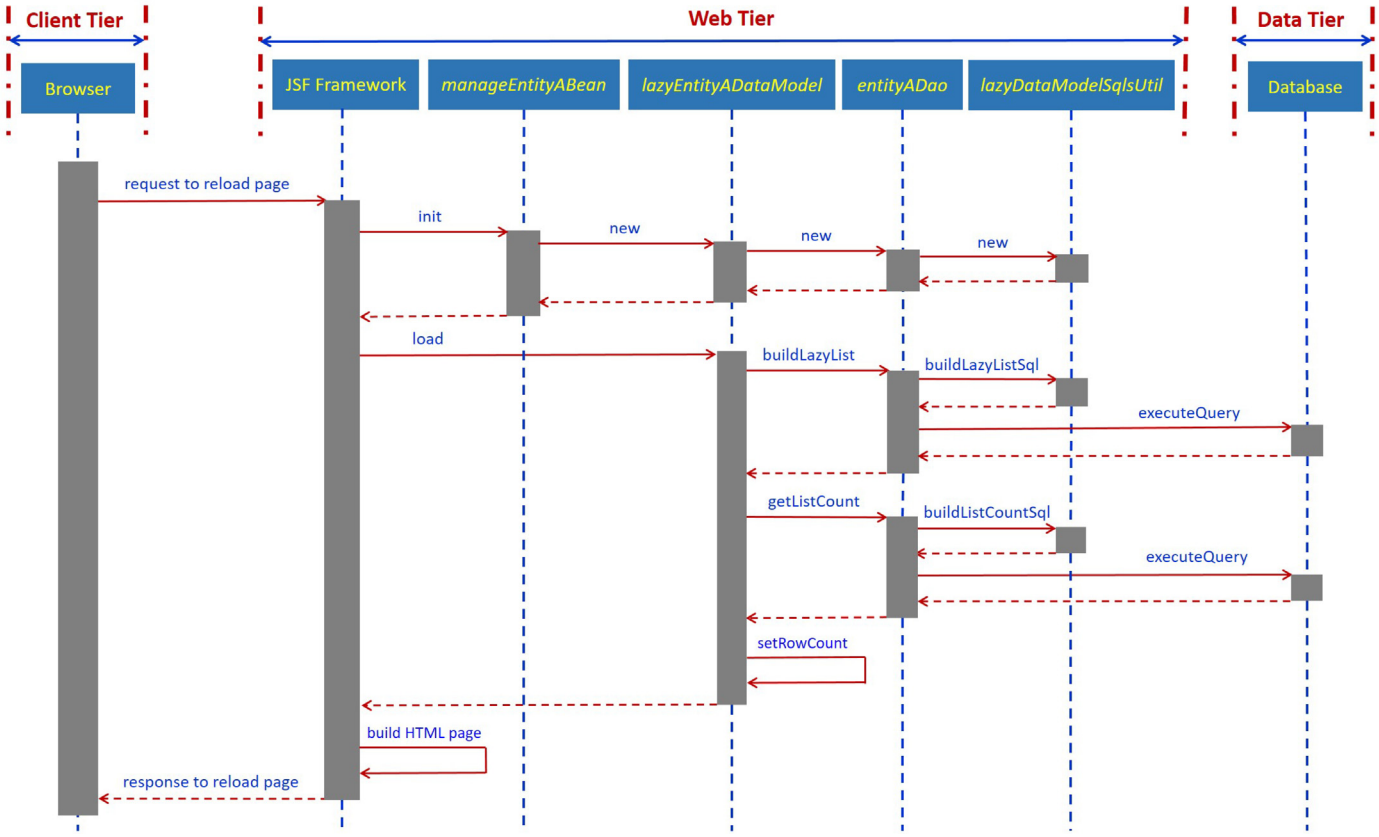

Fig. 12. A sequence diagram that illustrates objects interactions to display a web page that contains a data table with lazy loading capabilities.

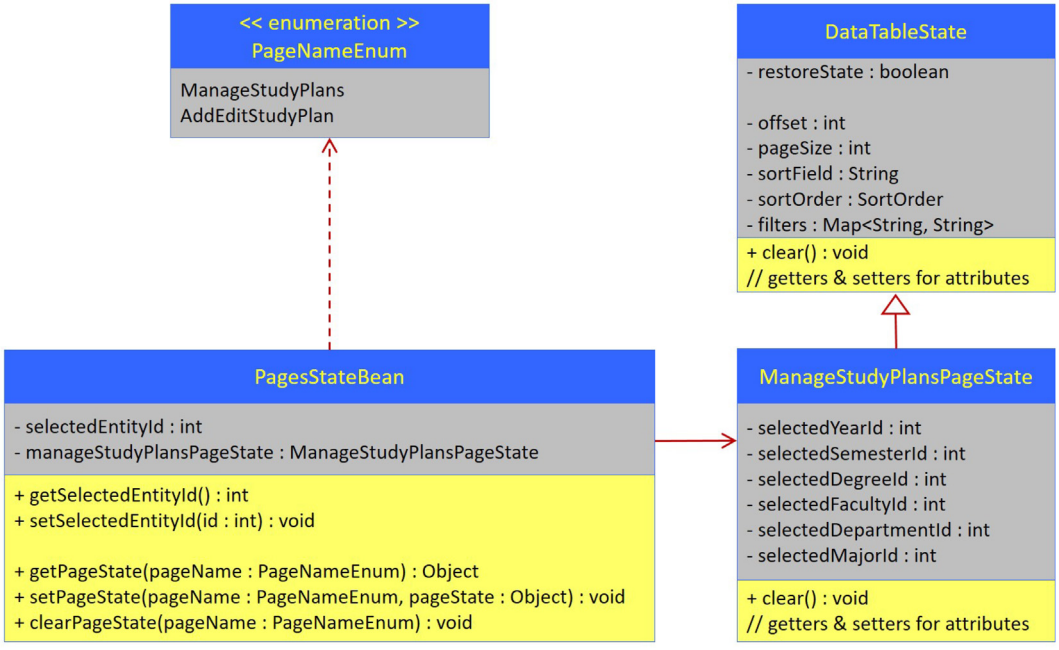

Fig. 13. A class diagram for the classes needed to restore a data management page state.

on the *pageState* instance to check whether to restore the state of the source page or not on load. The previous method usually returns *true* upon navigating from a related page back to the source page, otherwise it returns *false* when the source page is initially loaded. In case the *restoreState* value is *true*, then the *restorePageState* method gets called to restore the page state (e.g., filtering criteria, checkboxes choices, and data table selections) to its state before moving to the related page, and subsequently the *clearPageState* method is invoked on the *pagesStateBean* instance to clear the saved state of the source page. Secondly, when a

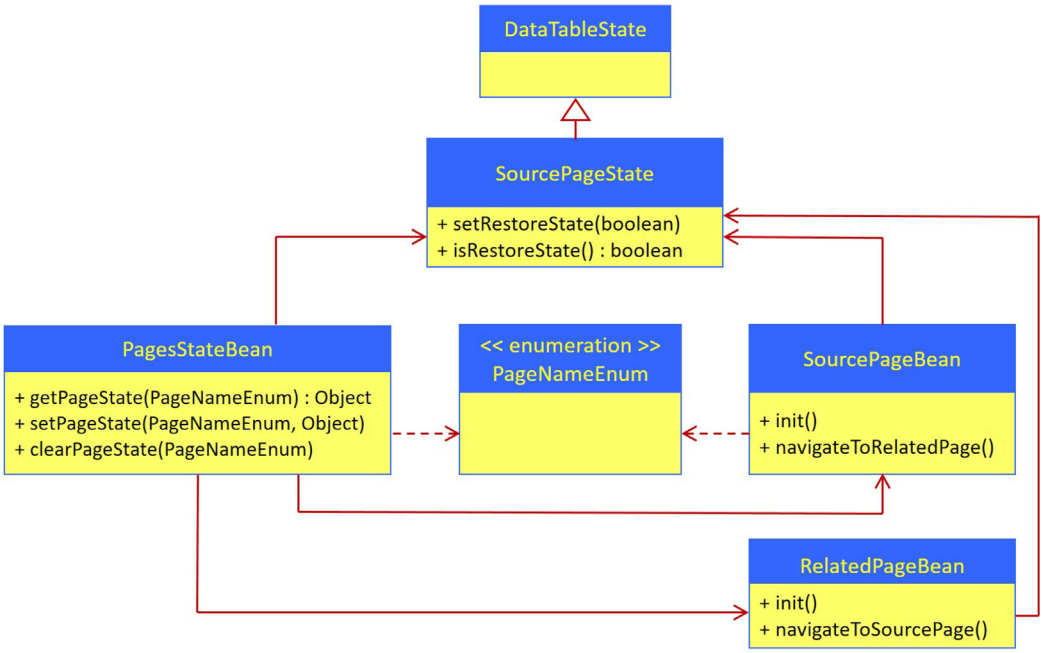

Fig. 14. A class diagram showing the classes needed to implement the page state restoration design pattern.

related page is requested (e.g., by clicking the Add button in Fig. 3), the *navigateToRelatedPage* method gets invoked on the *sourcePageBean* instance. The previous method next calls the *initSourcePageState* method to initialize an object of the source page state class with the current state of the page. The initialized object is then saved in the *pagesStateBean* via the *setPageState* method to make the source page state available across several pages. The JSF framework then navigates to the related page, and later invokes the init method on the *relatedPageBean* instance in which the stored source page state (i.e., pageState) is obtained via the *getPageState* method. Thirdly, when the source page is requested from the related page, the *navigateToSourcePage* method gets invoked on the *relatedPageBean* instance. Next, the *setRestoreState* method is called to set the value of the *restoreState* flag in the *pageState* instance to *true*. This flag is then used in the *init* method of the source page to determine whether to restore the page state or not on load as explained previously. The *pageState* object is later updated in the *pagesStateBean* via the *setPageState* method. The JSF framework then navigates to the source page, and later invokes the *init* method on the *sourcePageBean* instance in which the source page state (i.e., *pageState*) is obtained via the *getPageState* method to proceed as discussed earlier.

3.5. Add and edit page pattern

Usually a JSF page and a managed bean class are at least needed to implement a dynamic screen in a web application. But when comparing the screens to add a new item to a data table (e.g., see Fig. 16) and to edit a selected item in a data table (e.g., see Fig. 17), it was obvious that they have an identical layout and contain almost the same UI components. Based on that, it makes a lot of sense to utilize one JSF page instead of two pages as well as one managed bean class rather than two classes to implement the two previous operations. In that regard, the *Add and Edit Page* design pattern is adopted to allow cutting in half the number of basic software modules needed to develop functionality to add and edit items of the same type. Therefore, this design pattern promotes code reuse, which shortens development time and reduces code complexity.

The class diagram shown in Fig. 18 demonstrates the classes needed to implement this design pattern. The source page for data management is associated with a managed bean named *SourcePageBean*, whereas the related add/edit page is linked to a managed bean named *AddEditPageBean*. Both managed beans maintain a reference to the *SessionScoped PagesStateBean* object in which the identifier for the selected entity in the source page will be saved when needed. Besides, an *EntityDao* class is required to support methods to retrieve entity data from the database as well as insert/update entity data in the database. In that regard, the *addEditEntity* method is required to set the value of the *selectedEntityId* in the *PagesStateBean* to zero or to the identifier of the selected entity when adding a new entity or editing a selected entity, respectively. Hence, the *selectedEntityId* value is used to determine whether to open the page in add or edit mode. In turn, the *init* method in the *AddEditPageBean* class would retrieve the entity data from the database in case the value of the *selectedEntityId* is greater than zero which means that an item was selected for editing in the source page. Otherwise, nothing will be loaded and it will be assumed that the add/edit page is in the add entity mode when the *selectedEntityId* value is equal to zero. Whilst the *save* method in the *AddEditPageBean* class would invoke the *insertEntity* method or the *updateEntity* method on an *EntityDao* object when the page is in the add mode or edit mode, respectively.

The sequence diagram in Fig. 19 shows objects interactions when navigating to an add/edit page and upon saving an entity from that page. Firstly, when a request is submitted to add or edit an entity (e.g., *EntityX*) from a data management page (e.g., see Fig. 3), the JSF framework invokes the related *addEditEntityX* method on the managed bean instance that is associated with that page (e.g., *manageEntityXBean*). Then, the *setSelectedEntityId* method gets invoked on the *SessionScoped pagesStateBean* instance to make the selected entity identifier available across multiple

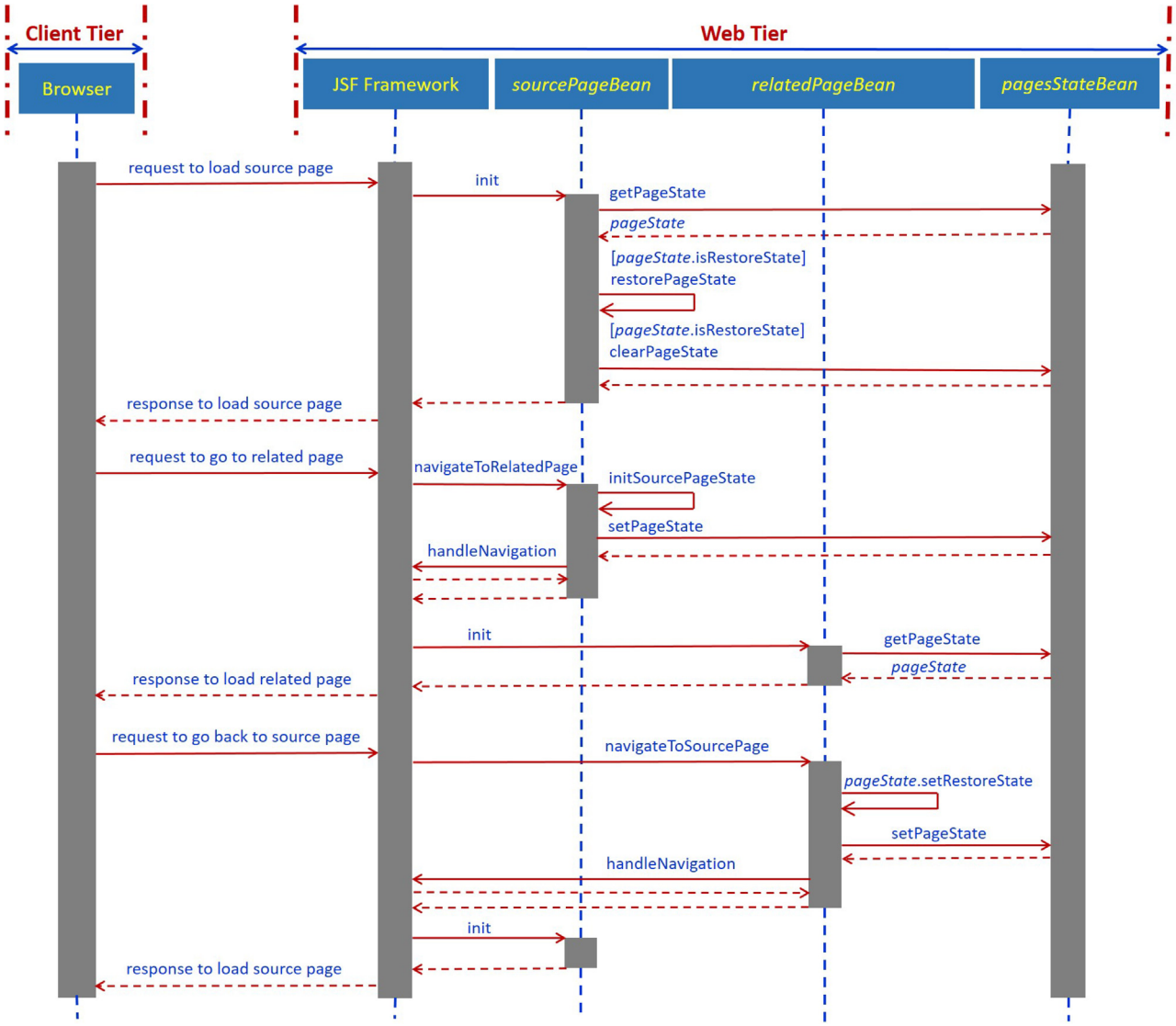

Fig. 15. A sequence diagram that shows objects interactions when loading a source page, navigating from a source page to a related page, and going back from the related page to its source page.

screens. The identifier value will be greater than zero in case an edit operation on a selected entity is performed. Whilst the identifier value will be set to zero to indicate that an operation to add a new entity is requested and thus no row selection is needed. Next, the *addEditEntityX* method instructs the JSF framework to navigate to the add/edit page afterwards. In turn, the JSF framework invokes the *init* method on the managed bean instance (e.g., *addEditEntityXBean*) of the add/edit page. The *init* method then invokes the *getSelectedEntityId* method on the *pagesStateBean* instance to get the previously saved identifier value. When the identifier value is equal to zero, the add/edit page is opened in add mode with all fields values set to their defaults. Otherwise, the page is assumed to be in edit mode and the *loadEntityX* method gets invoked to set the fields values according to those of the selected entity that were obtained from the database. Secondly, when the *Save* button in the add/edit screen (see Fig. 17) is clicked, a new entity will be inserted into the database if the page is in add mode. Whilst the values of the selected entity will be updated in the database in case the page is in edit mode. Next, the *selectedEntityId* value is reset in the *pagesStateBean* instance to start all over again if needed. Later, the JSF framework is instructed to navigate back to the source page.

4. Validation and results

The validation of the proposed design patterns in several web applications as well as comparison results to related work are discussed in the following subsections.

4.1. Design patterns validation

The utilization of the data management UI page design pattern was explored in the following web applications: SIS, HRS, AIS, online exams system, help desk system, and bus tracking system. Besides, the number of times the table data lazy loading design pattern was used in the found data management UI pages was also recorded. The investigation results are summarized in Table 1, in which it is obvious that both of the previous two design patterns were largely used in all these applications. Accordingly, the largest number of data management UI pages in all applications was 465 as discovered in the SIS, whereas only 7 of such pages were used in the Help Desk system. Noting that the more the data management UI page design pattern is used in an application, the larger and more complex the application. Similarly, the largest number

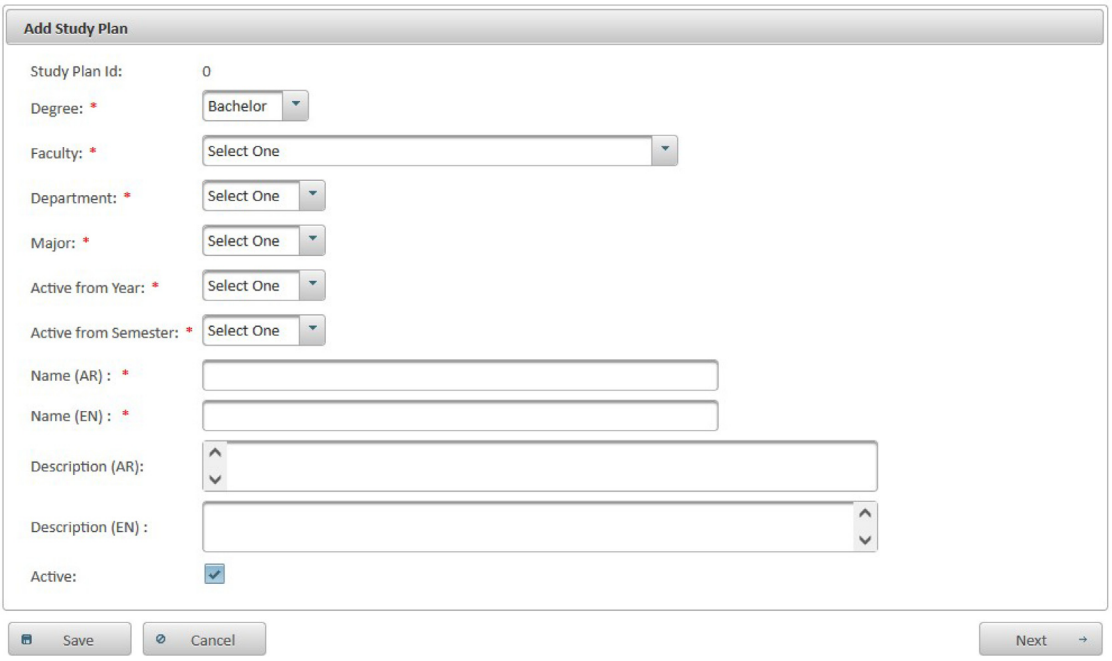

Fig. 16. A screen to add study plan header information with all fields values set according to their defaults.

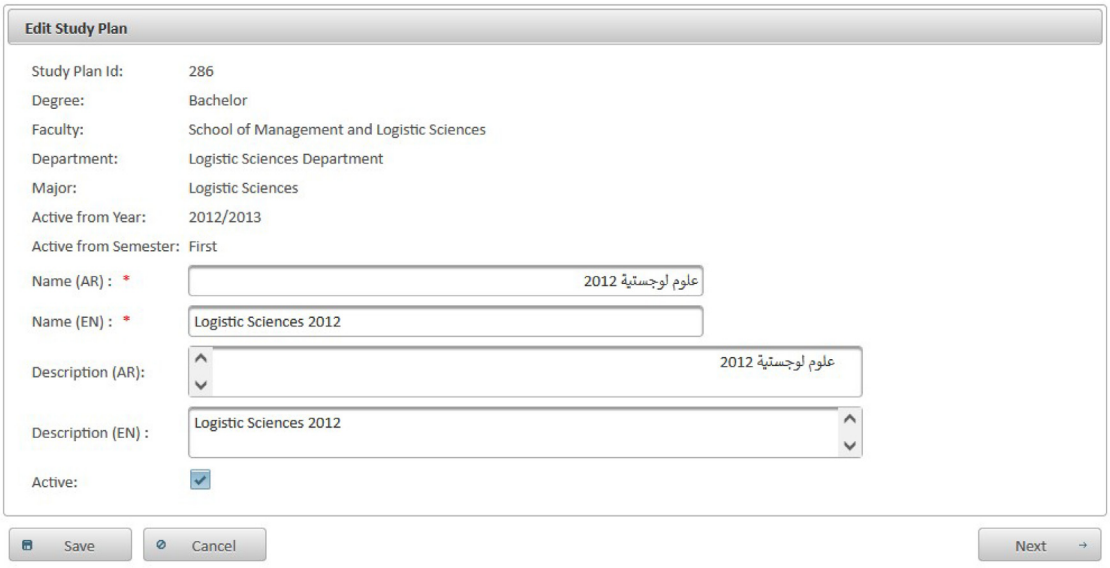

Fig. 17. A screen to edit study plan header information with all fields values initialized to those of the selected entity (row) in Fig. 3.

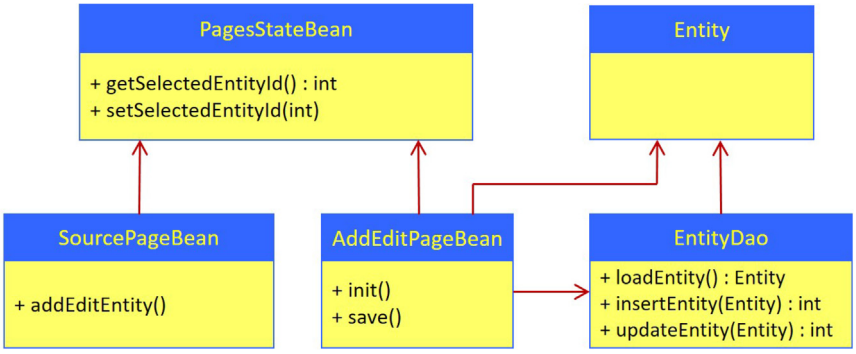

Fig. 18. A class diagram for the classes needed to implement the add and edit page design pattern.

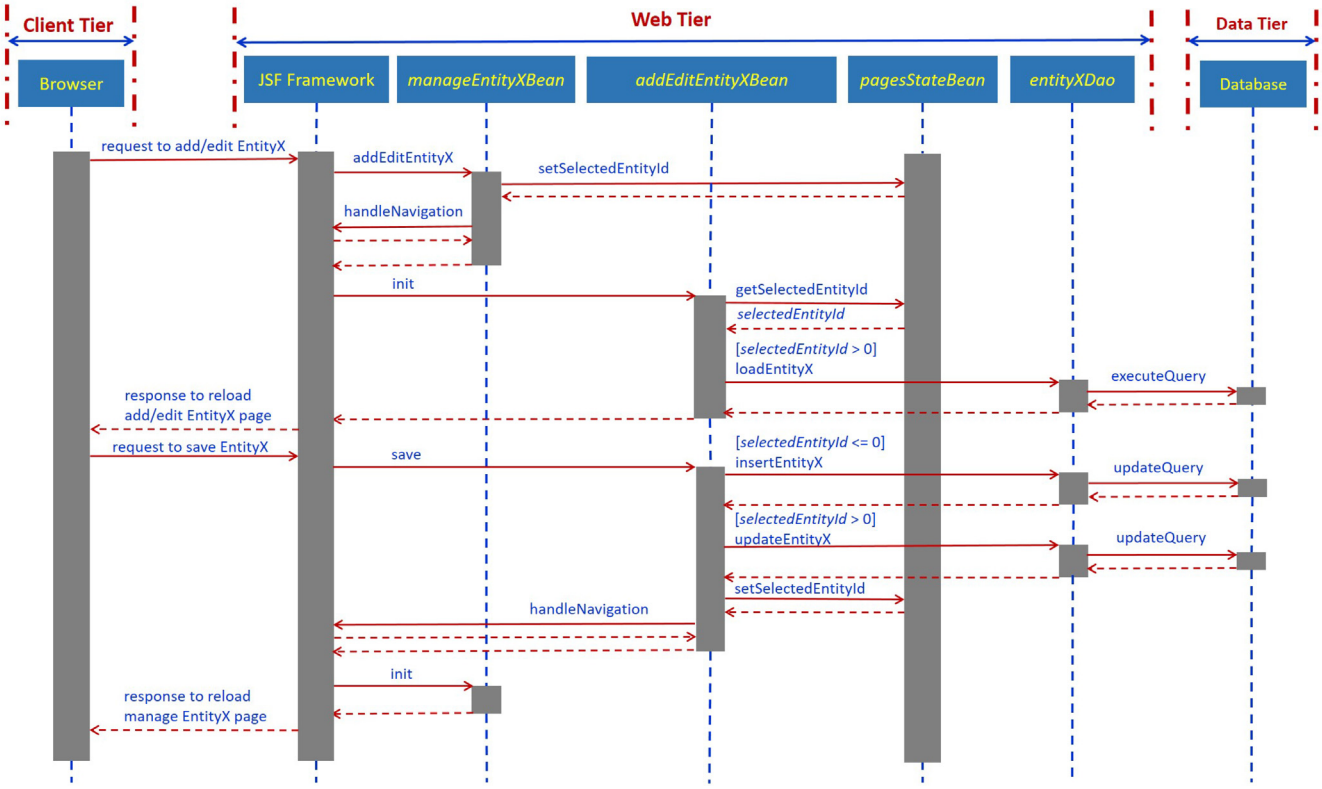

Fig. 19. A sequence diagram that illustrates objects interactions when navigating to an add/edit page and upon saving an entity from that page.

Table 1
The recurrence of the data management UI page as well as table data lazy loading design patterns in six different applications.

	# of Data Management UI Pages	# of Data Tables with Lazy Loading	% of Lazy to All Data Tables
SIS	465	85	18 %
HRS	163	28	17 %
AIS	61	15	25 %
Online Exams	15	4	27 %
Bus Tracking	13	3	23 %
Help Desk	7	2	29 %

of data tables with lazy loading capabilities in all systems was 85 as found in the SIS, while only 2 of such data tables were utilized in the relatively small Help Desk system. By analyzing the percentage of applying the table data lazy loading design pattern to the tables in the data management pages in the aforementioned systems, it was found that it ranged from 17 % in the HRS to 29 % in the Help Desk system as shown in Table 1. The fact that the computed percentages were not high is related to only applying the lazy loading design pattern to data tables that are expected to have large number of rows. This is suitable to avoid complicating the data table design especially when no obvious performance gains can be obtained by applying lazy loading to a table containing a small number of rows. In that regard, the positive impact of lazy loading on page load time was also examined when opening the manage admission applications page (Al-Hawari et al., 2017) in the SIS with, and without, lazy loading enabled. Given that the data table in that page could contain 20,472 rows and the table page size is set to 10 rows, the measured page load times were 2 s and 128 s when lazy loading was on and then off, respectively. Hence, lazy loading has drastically improved the page load time in this case.

Furthermore, the usage of the dependent dropdown filters design pattern was investigated in the SIS, which is considered the largest system amongst the aforementioned six systems. Based

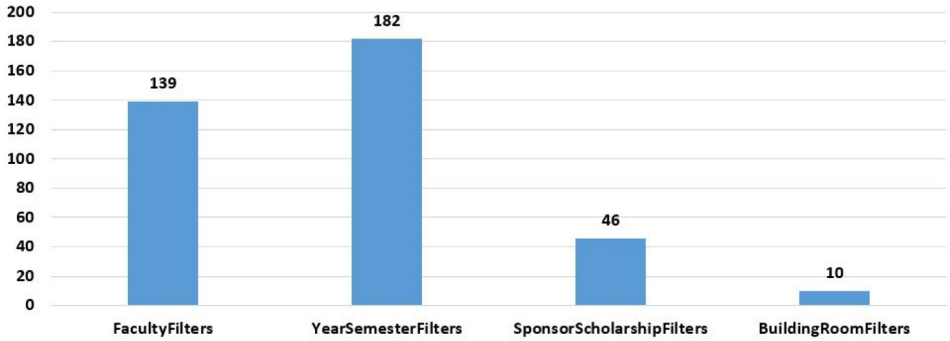

Fig. 20. The most used dependent dropdown filters in the SIS.

on the results shown in Fig. 20, the *FacultyFilters* utility class is reused in the implementation of 139 data management pages. Additionally, it was found that three other dependent dropdown filters were repeatedly utilized in that system. Firstly, the *YearSemesterFilters* class is used 182 times to search for data that depends on academic years and their related semesters. Secondly, the *SponsorScholarshipFilters* class is referenced 46 times to find data that depends on sponsors and their offered scholarships. Thirdly, the *BuildingRoomFilters* class is used 10 times to identify data that is related to buildings and their rooms.

In addition, the usage of the page state restoration design pattern was explored in the largest three systems of the previous six systems as shown in Fig. 21. Accordingly, this design pattern is used 157, 63, and 26 times in the SIS, HRS, and AIS applications, respectively. In that context, the number of help desk tickets to report issues related to the page state restoration design pattern over five years were also extracted as shown in Fig. 22. Such results were collected as they assert that users depend on that feature and find it useful. Last but not least, the usage of the add edit page design pattern was also tracked in the largest three systems as

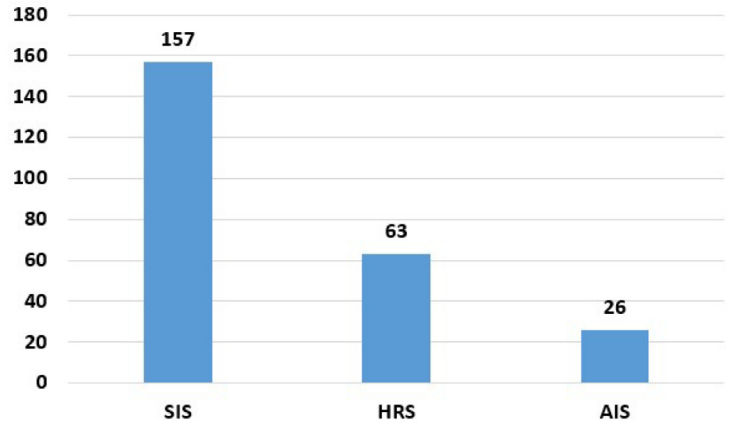

Fig. 21. The recurrence of the page state restoration design pattern in the SIS, HRS, and AIS applications.

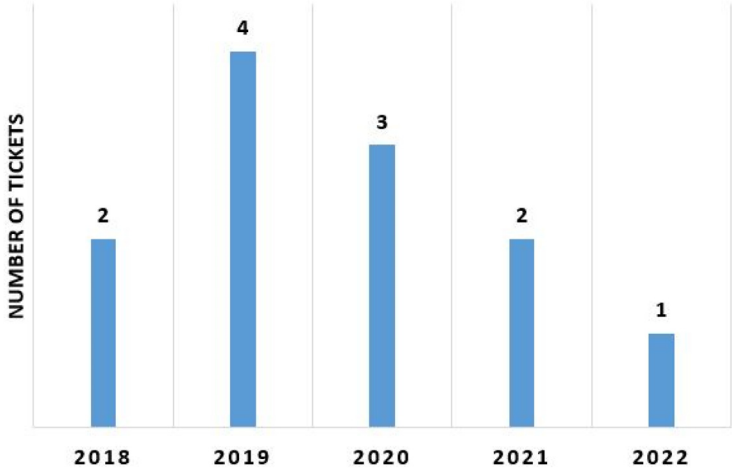

Fig. 22. The number of help desk tickets to report issues related to the page state restoration design pattern over five years.

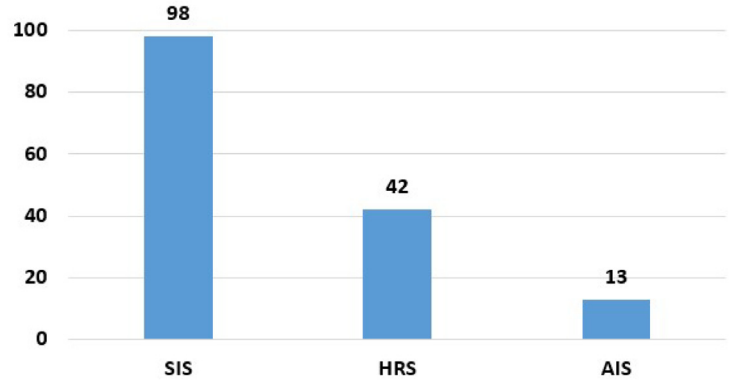

Fig. 23. The usage of the add edit page design pattern in the SIS, HRS, and AIS applications.

Table 2
The comparison results of this work to the most related studies.

	This Work	(Crawford & Kaplan, 2003)	(Heer & Agrawala, 2006)	(Fowler, 2012)	(Joshi, 2016)	(Bach et al., 2022)
Data Management UI Page	Yes	No	No	No	No	Page Layout & Data Information
Dependent Dropdown Filters	Yes	No	No	No	No	No
Table Data Lazy Loading	Yes	No	No	Lazy Load	Lazy Load	No
Page State Restoration	Yes	No	No	Server Session State	No	No
Add Edit Page	Yes	No	No	No	No	No

illustrated in Fig. 23. Based on that, this design pattern is utilized 98, 42, and 13 times in the SIS, HRS, and AIS systems, respectively.

4.2. Comparison to related work

The comparison results of this work to the most relevant studies are shown in Table 2. Accordingly, both the *Dependent Dropdown Filters* and *Add Edit Page* design patterns were not discussed by any of those studies. The work in Bach et al. (2022) introduced the *Data Information* and *Page Layout* design patterns to identify the types of information displayed in dashboards and to describe how widgets are arranged in a dashboard, respectively. The proposed *Data Management UI Page* pattern is similar to the previous two patterns in that it covers page layout and information, however it focuses on data management pages rather than dashboards. A *Lazy Load* design pattern was mentioned in Fowler (2012) and Joshi (2016), however it was generally described and not tied to *Table Data Lazy Loading* as proposed in this work. Not to mention, the *Server Session State* pattern mentioned in Fowler (2012) covered the need to preserve session state on the server side, but it did not address the mechanisms to restore the state of a data management page as reported in this study using the *Page State Restoration* pattern.

5. Conclusions

This paper discussed the design patterns for five features that recurred hundreds of times in the web-based information systems that have been developed by the GJU software development team. In that context, the outcomes of the *Data Management UI Page* design pattern are the availability of consistent UI pages as well as enabling system developers to implement such pages much faster. The *Dependent Dropdown Filters* pattern supports code reuse, which reduces software development time. The *Table Data Lazy Loading* pattern allows developing that feature in a simple and quick manner. The *Page State Restoration* as well as *Add Edit Page* patterns promote software user-friendliness and cut down the development times of the related features. Apparently, to the best of our knowledge, the *Dependent Dropdown Filters* and *Add Edit Page* patterns were not discussed in any of the related work.

As far as future work, the goal is to first introduce more features that recur many times in web-based applications. After that, suitable design patterns for such functions will be proposed to speed up the development time of the related modules as well as produce well-structured, comprehendible, and maintainable code.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Al-Hawari, F., 2017. Analysis and design of an accounting information system. Int. Res. J. Electron. Comput. Eng. 3 (2), 16–21.

Al-Hawari, F., Alufeishat, A., Alshawabkeh, M., Barham, H., Habahbeh, M., 2017. The software engineering of a three-tier web-based student information system (MyGJU). Comput. Appl. Eng. Educ. 25 (2), 242–263.

Al-Hawari, F., Al-Sammarraie, M., Al-Khaffaf, T., 2020. Design, validation, and comparative analysis of a private bus location tracking information system. J. Adv. Transp. 2020, 8895927. https://doi.org/10.1155/2020/8895927.

Al-Hawari, F., Barham, H., 2019. A machine learning based help desk system for IT service management. J. King Saud Univ.-Comput. Inf. Sci. https://doi.org/10.1016/j.jksuci.2019.04.001.

Al-Hawari, F., Alshawabkeh, M., Althawbih, H., Abu Nawas, O., 2019. Integrated and secure web-based examination management system. Comput. Appl. Eng. Educ. 27 (4), 994–1014.

Al-Hawari, F., Barham, H., Al-Sawaeer, O., Alshawabkeh, M., Alouneh, S., Daoud, M.I., Alazrai, R., 2021. Methods to achieve effective web-based learning management modules: MyGJU versus Moodle. PeerJ Comput. Sci. 7, e498.

Al-Hawari, F.H., Habahbeh, M.S., 2020. Secure and robust web services for e-payment of tuition fees. Int. J. Eng. Res. Technol. 13 (7), 1795–1801.

Ayeva, K., Kasampalis, S., 2018. Mastering Python design patterns: A guide to creating smart, efficient, and reusable software. Packt Publishing Ltd., Birmingham, United Kingdom.

Bach, B., Freeman, E., Abdul-Rahman, A., Turkay, C., Khan, S., Fan, Y., Chen, M., 2022. Dashboard design patterns. arXiv preprint. doi:https://doi.org/10.48550/arXiv.2205.00757.

Bloom, G., Alsulami, B., Nwafor, E., Bertolotti, I.C., 2018. Design patterns for the industrial Internet of Things. Paper presented at the 2018 14th IEEE International Workshop on Factory Communication Systems (WFCS).

Burns, B., Oppenheimer, D., 2016. Design patterns for container-based distributed systems. Paper presented at the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16).

Cooper, J.W., 2000. Java design patterns: a tutorial. Addison-Wesley Professional, Boston, MA, USA.

Crawford, W., Kaplan, J., 2003. J2EE design patterns: patterns in the real world. O'Reilly Media Inc, Newton, MA, USA.

Fernandes, S.M.M., Jose, P., 2017. Architectural pattern for native android applications. Revista de Sistemas e Computação-RSC 7 (2), 163–178.

Fowler, M., 2012. Patterns of enterprise application architecture. Addison-Wesley, Boston, MA, USA.

Guest, W., Wild, F., Di Mitri, D., Klemke, R., Karjalainen, J., Helin, K., 2019. Architecture and design patterns for distributed, scalable augmented reality and wearable technology systems. Paper presented at the 2019 IEEE International Conference on Engineering, Technology and Education (TALE).

Haq, M. S., Anwar, Z., Ahsan, A., Afzal, H., 2017. Design pattern for secure object oriented information systems development. Paper presented at the 2017 14th International Bhurban Conference on Applied Sciences and Technology (IBCAST).

Heer, J., Agrawala, M., 2006. Software design patterns for information visualization. IEEE Trans. Visual. Comput. Graph. 12 (5), 853–860.

Hussain, S., Keung, J., Sohail, M.K., Khan, A.A., Ilahi, M., 2019. Automated framework for classification and selection of software design patterns. Appl. Soft Comput. 75, 1–20.

Joshi, B., 2016. Beginning solid principles and design patterns for asp.net developers. Springer, New York, NY, USA.

JSF, 2022. Java Server Faces (JSF). Retrieved from http://www.javaserverfaces.org/.

Juneau, J., 2013. Introducing Java EE 7: a look at what's new. Apress, New York, NY, USA.

Lu, Q., Xu, X., Bandara, H. D., Chen, S., Zhu, L., 2021. Patterns for blockchain-based payment applications. Paper presented at the 26th European Conference on Pattern Languages of Programs.

Lu, Q., Zhu, L., Xu, X., Whittle, J., Douglas, D., Sanderson, C., 2022. Software engineering for responsible AI: An empirical study and operationalised patterns. Paper presented at the 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).

Malik, Z. H., Munir, T., Ali, M., 2020. UI design patterns for flight reservation websites. Paper presented at the Future of Information and Communication Conference.

Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Wagner, S., 2022. Software engineering for AI-based systems: a survey. ACM Trans. Software Eng. Methodol. 31 (2), 1–59.

Mayvan, B.B., Rasoolzadegan, A., Yazdi, Z.G., 2017. The state of the art on design patterns: A systematic mapping of the literature. J. Syst. Software 125, 93–118.

Mythily, M., Nambiar, R., Prabhavathy, K., Joseph, I.T., 2019. A design pattern structure specification to extract statistical report. Int. J. Innov. Technol. Exploring Eng. 8 (10), 3066–3070.

PrimeTek, 2022. PrimeFacses: Leading provider of open source UI component libraries. Retrieved from https://www.primefaces.org/showcase/.

Rajasekar, V., Sondhi, S., Saad, S., Mohammed, S., 2020. Emerging design patterns for blockchain applications. Paper presented at the ICSOFT.

Ritter, D., May, N., Rinderle-Ma, S., 2017. Patterns for emerging application integration scenarios: A survey. Inf. Syst. 67, 36–57.

Singh, J., Chowdhuri, S.R., Bethany, G., Gupta, M., 2021. Detecting design patterns: a hybrid approach based on graph matching and static analysis. Inf. Technol. Manage. 1–12. https://doi.org/10.1007/s10799-021-00339-3.

Sousa, T.B., Ferreira, H.S., Correia, F.F., 2021. A survey on the adoption of patterns for engineering software for the cloud. IEEE Trans. Software Eng. 48 (6), 2128–2140.

Tkaczyk, R., Wasielewska, K., Ganzha, M., Paprzycki, M., Pawlowski, W., Szmeja, P., Fortino, G., 2018. Cataloging design patterns for internet of things artifact integration. Paper presented at the 2018 IEEE International Conference on Communications Workshops (ICC Workshops).

Washizaki, H., Khomh, F., Guéhéneuc, Y.-G., Takeuchi, H., Natori, N., Doi, T., Okuda, S., 2022. Software-engineering design patterns for machine learning applications. Computer 55 (3), 30–39.

Wikipedia. (2022). Software design pattern. Retrieved from https://en.wikipedia.org/wiki/Software_design_pattern.