Department of Computer Science
Technical University of Cluj-Napoca

Software Design

Project documentation

Name: Mera Mihai
Group: 30434

Contents

Chapter 1

Requirement

The requirement for this project is:
Problema 26
Dezvoltați o aplicație care poate fi utilizată ca soft educațional pentru studiul cercului. Aplicația va avea 2 tipuri de utilizatori: elev și administrator. Utilizatorii de tip elev pot efectua următoarele operații fără autentificare:

- desenarea interactivă a cercurilor prin înlocuirea creionului și a riglei cu mouse-ul și alegerea stilului de desenare (culoare, stil linie, grosime linie);
- calcularea și afișarea unor proprietăți: aria unui cerc, lungimea unui cerc, aria unui sector de cerc, lungimea unui arc de cerc;
- vizualizarea unor cercuri particulare:
  - cercul circumscris unui poligon (dacă poligonul este inscriptibil)
  - cercul înscris (dacă poligonul este circumscriptibil)
  - cercuri specifice unui triunghi: cercul lui Tucker, cercurile lui Lucas, cercul ortocentroidal, cercurile lui Neuberg, cercul lui Adams, cercul lui Brocard
- salvarea/încărcarea unui cerc într-un/dintr-un fișier xml;
- solicitarea unui cont pentru testarea cunoștințelor

Utilizatorii de tip elev pot efectua următoarele operații după autentificare:

- verificarea cunoștințelor prin efectuarea unui test de 10 întrebări (alese aleator dintr-un set de 50 de întrebări) și vizualizarea punctajului obținut după finalizarea testului
- vizualizarea unei statistici după punctaj utilizând grafice (structură radială, structură inelară, etc.).

Utilizatorii de tip administrator pot efectua următoarele operații după autentificare:

- Operații CRUD pentru informațiile legate de utilizatorii aplicației care necesită autentificare;
- Vizualizarea listei tuturor utilizatorilor care necesită autentificare.

Chapter 2

Technologies

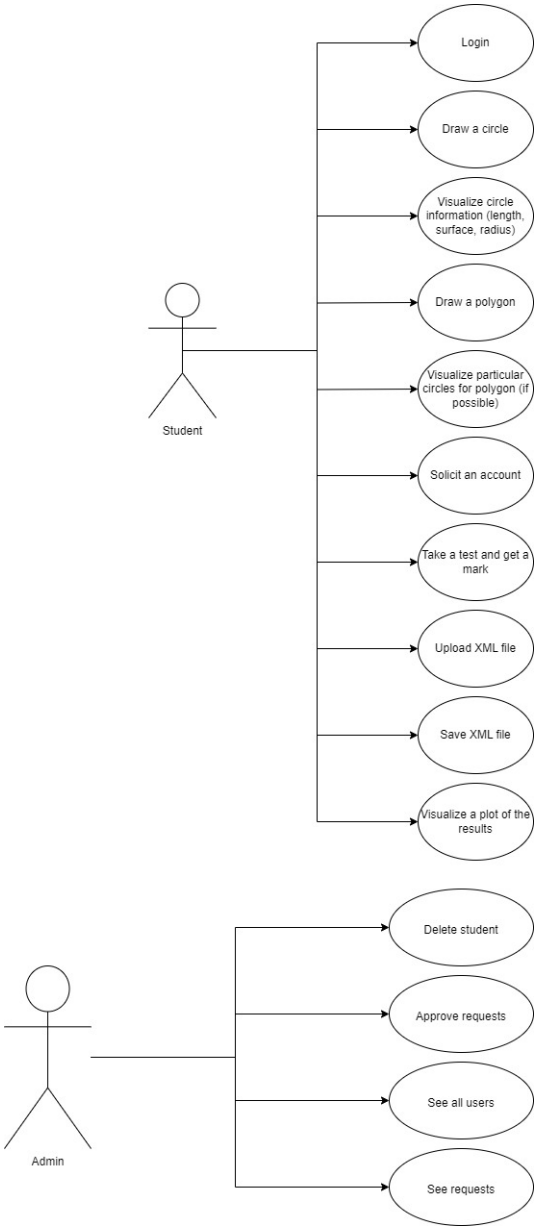
- Java Swing - Java Swing is a GUI (Graphical User Interface) widget toolkit for Java, which allows developers to create desktop applications with a rich graphical interface. It includes a variety of components such as buttons, text fields, labels, tables, and menus, which can be customized and arranged to create a user-friendly interface. Java Swing is part of the Java Foundation Classes (JFC) and provides a platform-independent API that can be used on different operating systems.
- Postgresql - Postgresql is a powerful open-source relational database management system (RDBMS) that is designed to handle large amounts of data and complex queries. It offers a wide range of features, including support for SQL (Structured Query Language), ACID (Atomicity, Consistency, Isolation, Durability) compliance, and concurrency control. Postgresql can be used for a variety of applications, including web applications, business intelligence, data warehousing, and geospatial data. It is known for its stability, reliability, and scalability, and is widely used in both small and large enterprises.
- Graphics2D - Graphics2D is a powerful 2D graphics API that is part of the Java 2D API. It allows developers to create and manipulate 2D shapes, lines, text, and images in Java. Graphics2D provides a wide range of features, including antialiasing, transformations, compositing, and text rendering, which can be used to create high-quality graphics and animations. Graphics2D is built on top of the Graphics class in Java, which provides basic 2D drawing operations. However, Graphics2D provides a more powerful and flexible API, with additional methods for creating complex shapes, handling transparency and compositing, and applying affine transformations. Graphics2D can be used to create a wide range of graphical applications, including games, scientific simulations, data visualization, and user interfaces.
- JAXB - The Java Architecture for XML Binding (JAXB) is a technology that enables Java developers to map Java classes to XML representations and vice versa. It provides a convenient way to convert XML documents to Java objects and back, which can be used in a variety of scenarios, including data binding, web services, and XML processing. JAXB includes a set of annotations that can be added to Java classes to specify how they should be mapped to XML documents. These annotations can be used to define element names, attribute names, and the order of elements in the XML document, among other things. JAXB also provides a set of APIs that can be used to marshall (convert Java objects to XML) and unmarshall (convert XML to Java objects) XML documents. These APIs make it easy to work with XML documents in Java applications and can be used with a variety of XML parsers.

- Apache POI - Apache POI (Poor Obfuscation Implementation) is a popular open-source Java library provided by the Apache Software Foundation. It allows developers to read, write, and manipulate Microsoft Office documents, including Excel spreadsheets (.xlsx and .xls formats). Apache POI provides comprehensive APIs to read data from existing Excel files and write data to new or existing Excel files. It supports both .xlsx (Excel 2007 and later) and .xls (Excel 97-2003) formats. You can easily work with individual cells in Excel spreadsheets using Apache POI. It allows you to read and modify cell values, apply formatting, and work with different data types such as numbers, dates, and strings.
- JFreeChart - JFreeChart is a popular open-source Java library for creating interactive and customizable charts and graphs. It provides a rich set of features and APIs for generating a wide range of chart types, including line charts, bar charts, pie charts, scatter plots, and more. JFreeChart supports a variety of chart types, allowing you to visually represent data in different formats. Some of the supported chart types include XY line charts, bar charts, stacked bar charts, area charts, pie charts, scatter plots, bubble charts, and Gantt charts. JFreeChart offers extensive customization options to tailor the appearance and behavior of charts to meet specific requirements. You can customize colors, fonts, axes, labels, legends, tooltips, plot backgrounds, and various other visual elements.
- FasterXML Jackson - FasterXML Jackson is a highly popular and feature-rich Java library designed for efficient JSON processing. It provides a comprehensive set of APIs and features that enable seamless serialization and deserialization of JSON data. Jackson offers a streaming API for handling large JSON files or processing data in a streaming fashion, as well as a tree model API for manipulating JSON data using a hierarchical approach. With its data binding capabilities, Jackson can automatically map JSON fields to corresponding Java object properties, simplifying the conversion process. The library also supports annotations for fine-grained customization, allowing developers to control the serialization and deserialization behavior. Jackson excels at handling complex scenarios, such as polymorphic types and integration with other Java frameworks like Spring and JAX-RS. Thanks to its performance optimizations and extensive feature set, FasterXML Jackson has become a go-to solution for JSON processing in Java applications, offering flexibility, speed, and robustness.

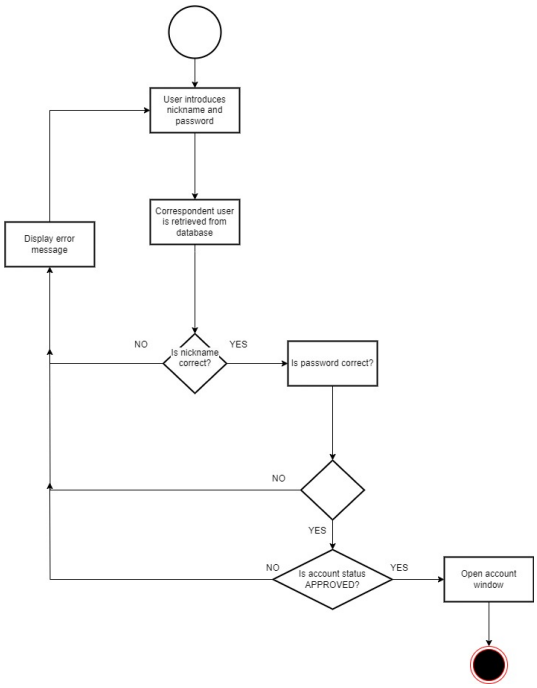
Chapter 3

Diagram description

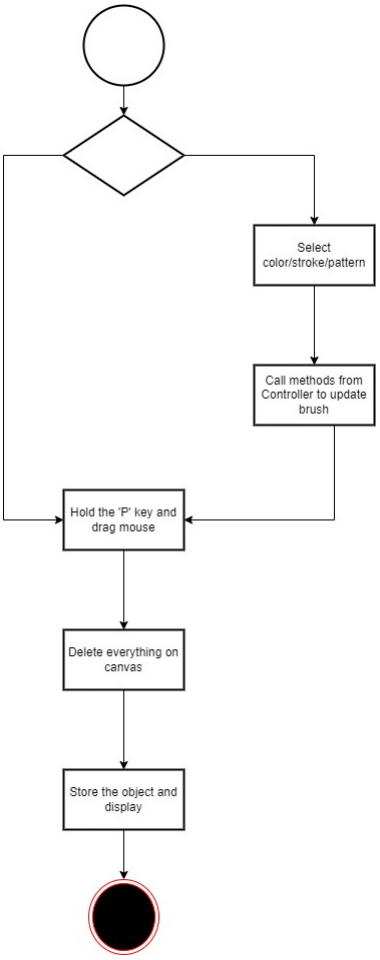
3.1 Use cases



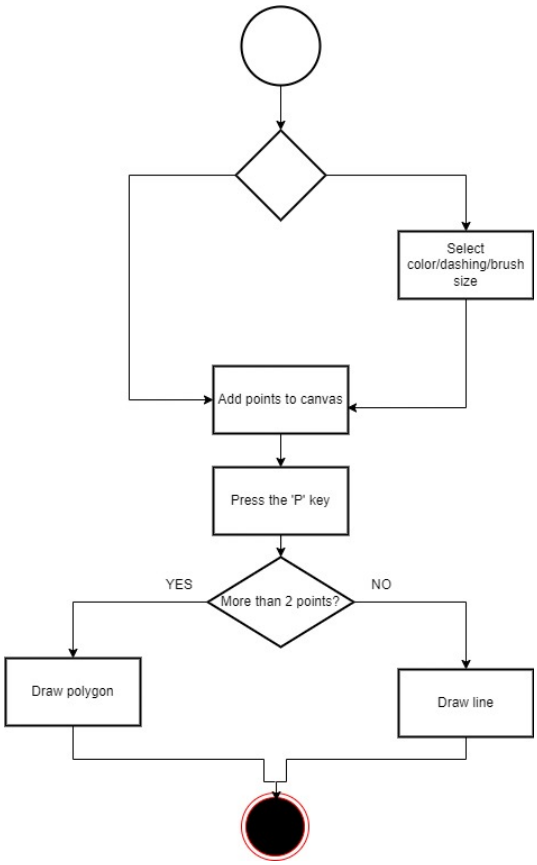
3.1.1 Login use-case



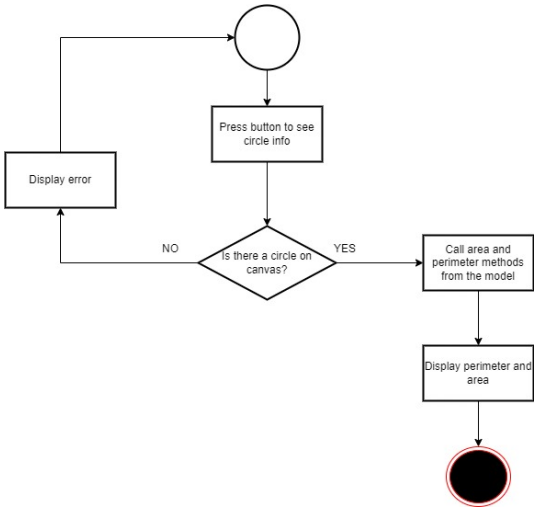
3.1.2 Draw circle use-case



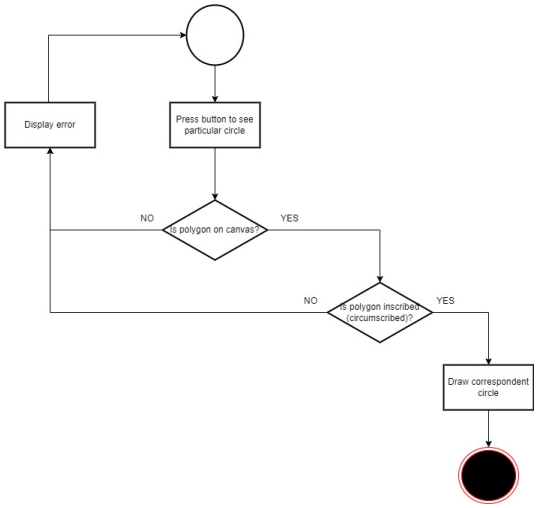
3.1.3 Draw polygon use-case



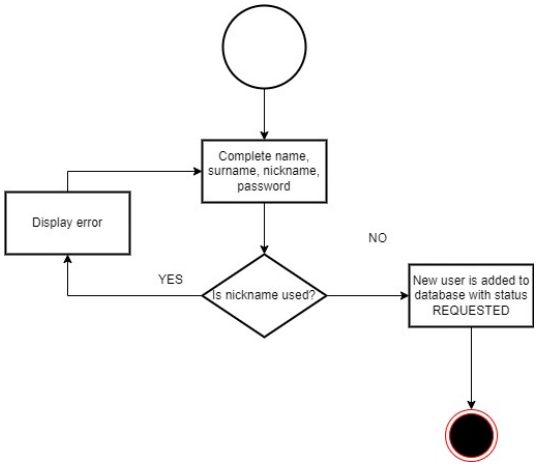
3.1.4 Circle data visualization use-case



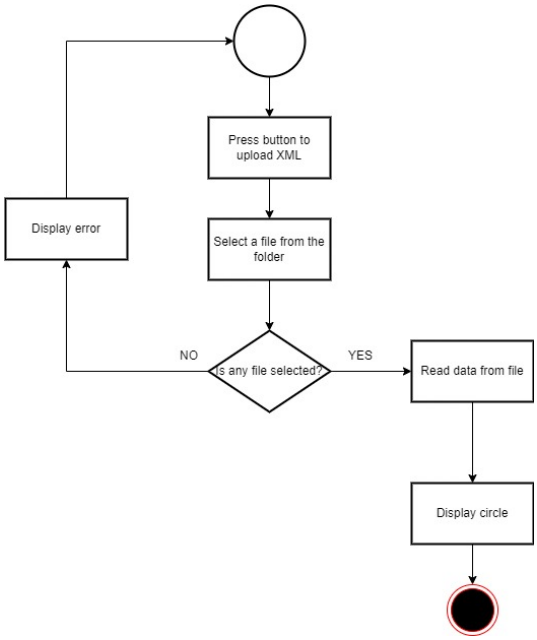
3.1.5 Particular circle visualization use-case



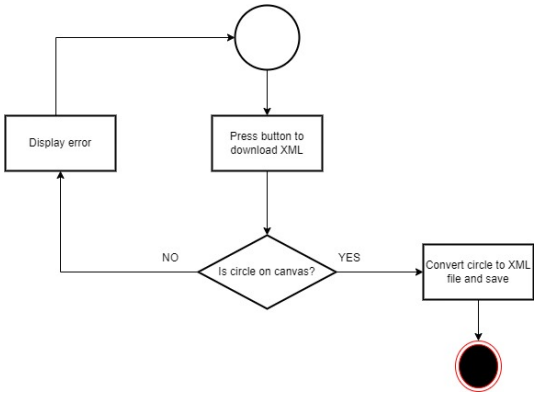
3.1.6 Account solicitation use-case



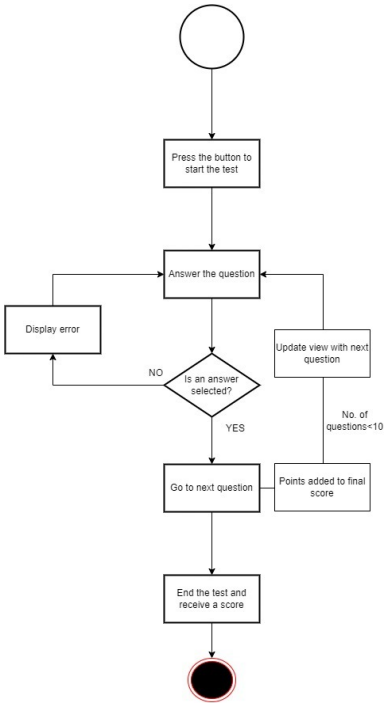
3.1.7 XML upload use-case



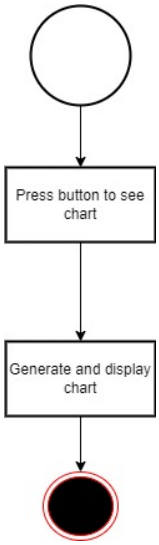
3.1.8 XML download use-case



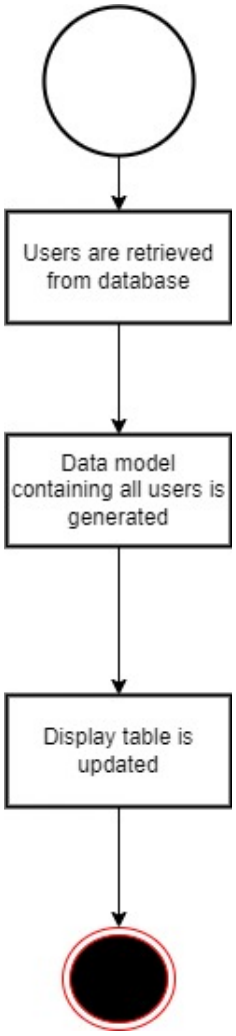
3.1.9 Test use-case



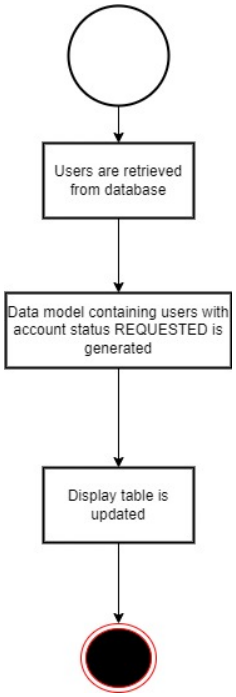
3.1.10 Plot visualization use-case



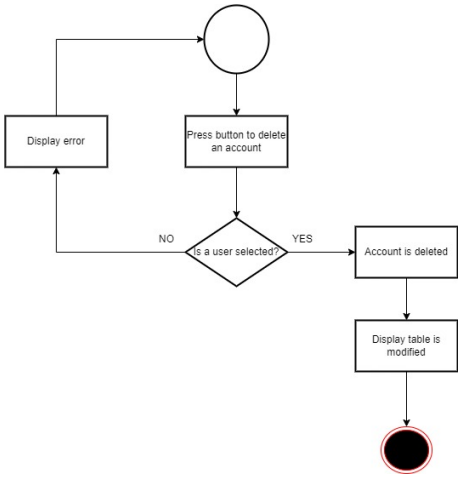
3.1.11 Admin users visualization use-case



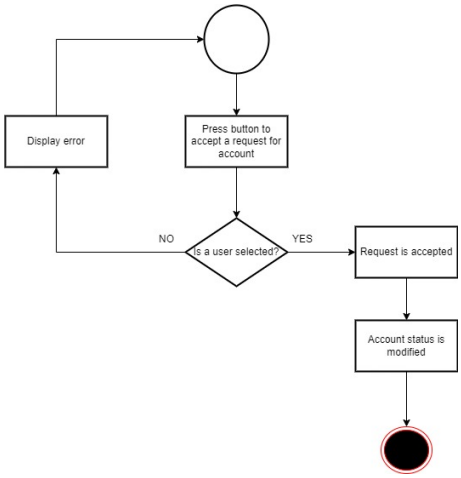
3.1.12 Admin requests visualization use-case



3.1.13 Admin delete use-case



3.1.14 Admin accept use-case



3.2 Model Geometry

The GeometryObject class is an interface that doesn't define any methods or fields. It's used as a common interface for all geometry objects in this codebase, meaning that any class implementing GeometryObject is considered a geometry object.
The Circle class represents a circle in two-dimensional space. It implements the GeometryObject interface and has two fields: origin and radius. The computeArea() method calculates the area of the circle using its radius and returns it, while the computePerimeter() method calculates the perimeter of the circle and returns it.
The Line class represents a straight line segment in two-dimensional space. It implements the GeometryObject interface and has two fields: start and end, which are both instances of the Point class. The computeSlope() method calculates the slope of the line segment, while the computeLength() method calculates the length of the line segment. The computeAngle() method takes another Line object as an argument and calculates the angle between the two lines. The computeMiddlePoint() method calculates the middle point of the line segment. Finally, the toString() method returns a string representation of the line segment in the format "startPoint - endPoint".
The Point class represents a point in two-dimensional space. It implements the GeometryObject interface and has two fields: x and y. The computeDistance() method takes another Point

object as an argument and calculates the distance between the two points. The toString() method returns a string representation of the point in the format "(x, y)".
The Polygon class represents a polygon in two-dimensional space. It implements the GeometryObject interface and has one field: vertices, which is an ArrayList of Point objects. The isInscribed() method returns a boolean indicating whether the polygon is inscribed, which is true if the polygon has exactly three vertices. Otherwise, it calculates the length and angle of each line segment in the polygon and returns false if any two line segments have different lengths or angles. The isCircumscribed() method returns a boolean indicating whether the polygon is circumscribed, which is true if all the vertices of the polygon lie on a circle. Finally, the getCircumcenter() method calculates the center of the circle that circumscribes the polygon by finding the intersection point of the perpendicular bisectors of any two sides of the polygon.

3.3 Model User

The Student class represents a student user in the system. It has five private instance variables: name, surname, nickname, password, and accountStatus. The first four variables store the student's personal information, while the accountStatus variable represents the status of the student's account, which is an instance of the AccountStatus enum. The Student class has two constructors, one that takes in five parameters to initialize all of the instance variables, and another that takes in four parameters and sets the accountStatus to REQUESTED by default. The class also provides getters and a setter for the name variable.
The AccountStatus enum is used to define the possible states of a user's account. In this case, the possible states are REQUESTED, APPROVED, and ADMIN. The REQUESTED state is assigned by default when a new Student object is created, and it means that the user has requested to join the platform, but their account hasn't been approved yet. The APPROVED state means that the user's account has been approved, and they can now log in and use the platform. The ADMIN state is used to denote the account of an administrator who has special privileges on the platform.

3.4 Model Quiz

Difficulty enum is an enumeration that represents the difficulty level of a quiz question. It has three possible values: EASY, MEDIUM, and HARD, each with an associated integer value representing the difficulty level.
Question class is a class that represents a quiz question. It has four instance variables: a String representing the question text, a Difficulty representing the question difficulty level, an ArrayList of Strings representing the answer options, and a String representing the file name of an optional image associated with the question. It has a constructor that takes these four parameters and sets them as instance variables. It also has methods to get each instance variable, as well as a method to randomize the order of the answer options and return the index of the correct answer.
Test class is a class that represents a quiz test. It has three instance variables: an ArrayList of Questions representing the quiz questions, an integer representing the maximum number of points possible, and an ArrayList of integers representing the indices of the correct answer for each question. It has a constructor that generates a random set of 10 questions from a question database, computes the maximum number of points possible based on the difficulty level of each question, and shuffles the answer options for each question. It also has methods to get each instance variable.
TestTableEntry class is a class that represents an entry in a quiz test results table. It has five

instance variables: an integer representing the index of the entry, two Strings representing the name and surname of the test taker, an integer representing the number of points earned on the test, and a Timestamp representing the time the test was taken. It has a constructor that takes these five parameters and sets them as instance variables. It also has methods to get each instance variable.

Chapter 4

Implementation details

4.1 Resources

DrawingCanvas extends the Canvas class in Java's AWT package. It represents a canvas for drawing and allows the user to draw points and circles. The DrawingCanvas class also handles user interactions, such as mouse clicks and key presses, to create and draw points and circles on the canvas.
The class has several fields, such as point, circle, containsPolygon, containsCircle, x, y, startX, startY, endX, and endY, which are used to keep track of the state of the canvas and the drawn shapes. The color and stroke fields represent the current color and stroke used for drawing.
The class has several public methods, such as displayError, setContainsPolygon, setContainsCircle, repaint, getColor, setColor, getStroke, and setStroke, which allow the user to interact with the canvas and modify its properties.
The Palette class is an enumeration that represents a color palette with various shades of blue. Each color in the palette is represented as an enum constant with a corresponding hexadecimal color code. The constructor of the enum takes a string parameter which is the hexadecimal representation of the color. The color() method returns a java.awt.Color object that corresponds to the enum constant's color. The class is useful for providing a set of pre-defined colors for a graphical user interface or for other visual applications.

4.2 Database

The class named "DatabaseCreator" is located in the package "repo". It has several methods for creating and filling tables in a database using SQL queries, as well as checking for existing tables.
The class contains four public methods: "createTableStudent()", "fillTableQuestions(String filename)", "createTestsTable()", and "insertAdmin()".
The method "createTableStudent()" creates a table named "STUDENTS" if it doesn't already exist. The table has columns for student name, surname, nickname, password, and account status. It returns a boolean value indicating whether the table was successfully created and an admin user was inserted.
The method "fillTableQuestions(String filename)" reads question data from a file and populates a table named "QUESTIONS". The table has columns for the question, its difficulty, and four possible answers, as well as an optional image file. It returns a boolean value indicating whether the table was successfully created and filled with data.
The method "createTestsTable()" creates a table named "TESTS" if it doesn't already exist. The table has columns for the student ID, test score, and the time the test was taken. It returns a boolean value indicating whether the table was successfully created.

The method "insertAdmin()" inserts a default admin user into the "STUDENTS" table, with a username, password, and account status of "ADMIN". It returns a boolean value indicating whether the insertion was successful.
The class uses another class named "Persistence" for handling the database connection and SQL queries. It also uses a class named "QuestionsPersistence" for inserting question data into the "QUESTIONS" table.

The enum class named DBConnectionInfo contains the details necessary for establishing a database connection. It has four constants named DRIVER, URL, USER, and PASSWORD, each representing the driver class name, the database URL, the username and password required for accessing the database, respectively. The constants are assigned their respective values in their constructors, which take a single String parameter representing the value to be assigned. The class also has a method getValue() which returns the value of the constant. This class is typically used in conjunction with a JDBC driver to establish a connection to a PostgreSQL database.

The Persistence class is used to connect to a PostgreSQL database and execute SQL queries.
The class has a constructor that tries to load the PostgreSQL JDBC driver using the value of the DRIVER constant defined in the DBConnectionInfo enum.
The class has a method called "connect()" that creates a connection to the PostgreSQL database using the values of the URL, USER, and PASSWORD constants defined in the DBConnectionInfo enum.
The class has a method called "close()" that closes the connection to the PostgreSQL database.
The class has a method called "executeQuery(String query)" that takes an SQL query as an argument, connects to the database using the "connect()" method, executes the query using a Statement object, and returns a boolean value indicating whether the query was successful or not.
The class has a method called "getDataTable(String query)" that takes an SQL query as an argument, connects to the database using the "connect()" method, executes the query using a Statement object, and returns a ResultSet object containing the result of the query.
Both the "executeQuery()" and "getDataTable()" methods use the "connect()" and "close()" methods to ensure that the connection to the database is properly established and closed after each query.

4.3 Design Patterns

- Proxy - In the context of client-server applications and image transmission, the Proxy Design Pattern, which is a structural design pattern, can be leveraged to optimize the handling and delivery of images. The Proxy pattern introduces an intermediary object, the proxy, which acts as a surrogate for the actual image object. The proxy serves as a gateway between the client and the server, allowing for controlled and efficient image transmission. The application uses this DP to send the images for the quiz to the client.
- Observer - The Observer design pattern is a behavioral pattern that establishes a one-to-many dependency between objects, such that when one object (the subject) changes its state, all dependent objects (observers) are automatically notified and updated. In Java, the Observer design pattern is commonly implemented using listeners. Listeners are interfaces that define callback methods to be executed when specific events occur. By implementing these listener interfaces, objects can register themselves as observers to be notified when events of interest happen. In Java, the java.util.Observer interface and

java.util.Observable class provide a basic implementation of the Observer pattern. The Observable class represents the subject and maintains a list of registered observers. It provides methods to add and remove observers, as well as to notify observers of state changes using the notifyObservers() method.

- Singleton - The Singleton Design Pattern is a creational pattern that guarantees a class has only one instance and provides global access to it. It restricts instantiation to a single object by defining a private constructor and offering a static method to access the instance. The Singleton pattern is beneficial when a single, consistent instance needs to be shared across multiple components, such as managing shared resources. It promotes efficiency, centralizes control, and simplifies access. However, caution should be exercised to avoid excessive coupling and ensure thread-safety. Overall, the Singleton pattern offers a way to create a single, globally accessible instance, ensuring consistency and efficient object management in an application. In the context of this application, it is used for creating a unique connection to the database, to avoid multiple instances of an object with the same purpose.
- Adapter - The Adapter Design Pattern plays a crucial role in bridging the gap between the different interfaces and behaviors of these input devices, being one of the most used structural design patterns. The Adapter pattern allows for the seamless integration of keyboard and mouse functionality into a Java application by providing a consistent and unified interface. By implementing adapter classes, the specific events and actions triggered by keyboards and mice can be translated and mapped to a common set of methods and events defined by the adapter interface. This enables the application to handle input from keyboards and mice without needing to directly deal with the intricacies of each device's unique event system. The Adapter pattern promotes code reusability and maintainability by decoupling the application logic from the underlying input devices. It simplifies the development process by providing a standardized interface that can be easily extended to support new input devices or modified to accommodate changes in existing devices.
- Marker - The Marker Design Pattern, also known as the Marker Interface Pattern, is a behavioral design pattern that involves the use of empty marker interfaces to add metadata or mark certain classes or objects with specific characteristics. The marker interfaces themselves do not contain any methods or fields; they simply act as a tag or flag to indicate that a class or object possesses certain traits or behaviors. By implementing a marker interface, a class can signal its participation in a particular category or group, allowing other components to identify and handle it accordingly. Marker interfaces can be used for various purposes, such as indicating serialization eligibility, enabling runtime reflection, denoting specific capabilities or features, or facilitating custom framework behavior. The Marker pattern promotes extensibility and flexibility by providing a mechanism to add additional functionality or behavior to classes without modifying their structure. However, it is important to use marker interfaces judiciously and ensure they have a clear purpose and meaningful impact on the system design.

4.4 Client-Server architecture

4.4.1 Client implementation

At a high level, the MVC pattern is designed to help separate the concerns of a GUI application into three distinct components: the Model, the View, and the Controller. Each of these

components has a specific role to play in the overall design of the application.

Here's a breakdown of what each of the components of MVC is responsible for:

Model
The Model is responsible for representing the underlying data and business logic of the application. It might contain things like database access code, web service calls, or other data-related operations. Essentially, the Model is where all the "real work" of the application happens.

View
The View is responsible for presenting the user interface to the user. It is typically implemented using some combination of HTML, CSS, and/or other graphical technologies. The View knows how to display data to the user, and how to respond to user input.

Controller
The controller in the MVC (Model-View-Controller) pattern acts as the intermediary between the model and the view components. Its primary responsibility is to handle user input, process it, and update the model or view accordingly.
The controller receives user input from the view component and determines the appropriate actions to take based on that input. It invokes the necessary methods in the model to perform data manipulation or retrieval operations. Once the model updates, the controller then updates the view component to reflect the changes.
The controller plays a crucial role in maintaining the separation of concerns in MVC. It encapsulates the application logic and orchestrates the flow of data between the model and the view. By decoupling the user interface from the data and business logic, the controller enables modular development, code reusability, and easier maintenance.

The Observer design pattern is a behavioral pattern that establishes a one-to-many dependency between objects, such that when one object (the subject) changes its state, all dependent objects (observers) are automatically notified and updated.
In Java, the Observer design pattern is commonly implemented using listeners. Listeners are interfaces that define callback methods to be executed when specific events occur. By implementing these listener interfaces, objects can register themselves as observers to be notified when events of interest happen.
In Java, the java.util.Observer interface and java.util.Observable class provide a basic implementation of the Observer pattern. The Observable class represents the subject and maintains a list of registered observers. It provides methods to add and remove observers, as well as to notify observers of state changes using the notifyObservers() method.
Observers in Java typically implement the java.util.Observer interface. This interface defines the update() method, which is called by the subject (Observable) to notify the observer of the state change. The update() method typically receives the updated data or a reference to the subject itself.
By using listeners and the Observer pattern in Java, objects can establish loose coupling and enable event-driven communication. This approach allows for extensibility and flexibility, as new observers can be added without modifying the subject or other existing observers.

The Client part of the application is MVC-based.

AccountController

The AccountController class is a controller component in the MVC pattern. It handles user actions and updates the associated AccountView accordingly. The class implements the ComponentAdapter and ActionListener interfaces. It has a reference to an AccountView object and a Language object. The goBack() method navigates back to the previous view. The retrieveTests() method retrieves tests from the database and updates the model in the view. The takeTest() method initiates the process of taking a test by creating a TestView object. The actionPerformed() method handles different action commands triggered by user interactions. The componentShown() method is triggered when the component is shown and updates the view. The generateChart() method creates a chart using JFreeChart library based on the test data in the model and displays it in a ChartFrame.

AdminController

The AdminController class is responsible for controlling the behavior of the AdminView in an MVC architecture. It implements the ActionListener and ListSelectionListener interfaces to handle user actions and list selection events, respectively.
The class has a reference to an AdminView object and a Language object for language-related functionality. It also maintains the index of the selected row in the table.
The actionPerformed() method handles different action commands triggered by user interactions. Depending on the command, it performs actions such as deleting a student, approving a student's request, navigating back, and displaying requested or all students.

DrawingCanvasController

The DrawingCanvasController class is responsible for controlling the behavior of the DrawingCanvas in an application that involves drawing geometric objects. It manages the collection of GeometryObject instances and interacts with the DrawingCanvas view.
Key features and responsibilities of the DrawingCanvasController class include:
The class maintains a reference to a DrawingCanvas object and an ArrayList called geometryObjects that stores the geometric objects drawn on the canvas.
It also has references to the DrawingCanvasMouseController and DrawingCanvasKeyController objects, which handle mouse and key events on the canvas.
The getGeometryObjects() and setGeometryObjects() methods provide access to the collection of geometric objects.

DrawingCanvasKeyController

The DrawingCanvasKeyController class is responsible for handling key events on the DrawingCanvas in an application that involves drawing geometric objects. It works in conjunction with the DrawingCanvasController to modify the canvas and add objects based on key input.
Key features and responsibilities of the DrawingCanvasKeyController class include:
The class extends the KeyAdapter class to override key event handling methods.
It maintains references to the DrawingCanvas and DrawingCanvasController objects.
The keyPressed() method is triggered when a key is pressed. If the Shift key is pressed, it sets the canvas to draw circles instead of points.

The keyReleased() method is triggered when a key is released. If the Shift key is no longer pressed, it sets the canvas to draw points again instead of circles.
The keyTyped() method is triggered when a key is typed. If the key is 'p', it attempts to create a polygon based on the points drawn on the canvas.

DrawingCanvasMouseController

The DrawingCanvasMouseController class is responsible for handling mouse events on the DrawingCanvas in an application that involves drawing geometric objects. It works in conjunction with the DrawingCanvasController to modify the canvas and add objects based on mouse input.
Key features and responsibilities of the DrawingCanvasMouseController class include:
The class extends the MouseAdapter class to override mouse event handling methods.
It maintains references to the DrawingCanvas and DrawingCanvasController objects.
The mouseClicked() method is triggered when the mouse is clicked. It checks if the canvas is not set to draw circles. If so, it sets the canvas to draw points, records the coordinates of the click, and performs necessary actions based on the current canvas state (e.g., clearing the canvas if a polygon or circle is already drawn).
The mousePressed() method is triggered when a mouse button is pressed. If the canvas is set to draw circles, it records the starting coordinates of the circle and prepares the canvas by clearing existing objects.
The mouseReleased() method is triggered when a mouse button is released. If the canvas is set to draw circles, it records the ending coordinates of the circle and triggers the canvas to repaint, displaying the newly drawn circle.

LoginController

The LoginController class is responsible for handling user interactions and actions related to the login functionality in an application. It is associated with the LoginView, which provides the user interface for login.
Key features and responsibilities of the LoginController class include:
The class implements the ActionListener interface to handle action events.
It maintains a reference to the LoginView object and a Language object.
The goBack() method is invoked when the user clicks the "BACK" button. It makes the canvas view visible and closes the login view.
The login() method is triggered when the user clicks the "LOGIN" button. It retrieves the entered nickname and password from the login view.
It performs validation to ensure that both the nickname and password are provided. If not, it displays an error message and resets the input fields.

NewAccountController

The NewAccountController class handles user interactions and actions related to creating a new account in an application. It is associated with the NewAccountView, which provides the user interface for creating a new account.
Key features and responsibilities of the NewAccountController class include:
The class implements the ActionListener interface to handle action events.
It maintains a reference to the NewAccountView object and a Language object.

The actionPerformed() method is invoked when an action event occurs, such as clicking a button. It checks the action command of the event and calls the corresponding method (request()) if the action is "REQUEST".
The request() method is triggered when the user clicks the "REQUEST" button to create a new account.
It retrieves the entered name, surname, nickname, and password from the NewAccountView.
The reset() method clears the name, surname, nickname, and password input fields in the new account view.

TestController

The TestController class is responsible for handling user interactions and actions related to taking a test in an application. It is associated with the TestView, which provides the user interface for the test-taking process.
Key features and responsibilities of the TestController class include:
The class implements the ActionListener interface to handle action events.
It maintains references to the TestView object, the current question index, the total points earned, the Test object containing the questions and correct answers, and a Language object.
The actionPerformed() method is invoked when an action event occurs, such as clicking a button. It checks the action command of the event and calls the corresponding method (start(), next(), or end()) based on the command.
The start() method is triggered when the user clicks the "START" button to begin the test. It initializes the index and points variables, creates a new Test object, and proceeds to the next question.
The next() method is called when the user clicks the "NEXT" button to move to the next question. It checks the user's answer for the current question, updates the points if the answer is correct, and displays the next question in the TestView. It also handles the visibility of the "NEXT" and "END" buttons based on the current question index.
The checkQuestion() method checks the user's selected answer for the current question and returns the corresponding option index (0, 1, 2, or 3).
The end() method is triggered when the user clicks the "END" button to finish the test. It performs the same actions as the next() method but also displays the test results in a message dialog. It calculates the points earned, constructs a result message using the Language object, displays the message, closes the TestView, and persists the test results using the TestPersistence class.
The reset() method clears the input fields in the test view. The class ensures that the user selects an option for each question and provides error messages if an option is not selected.

4.4.2 Server implementation

SOA

Service-Oriented Architecture (SOA) is an architectural style that promotes loose coupling, reusability, and interoperability of software components within a system. It is a design approach that emphasizes the use of self-contained and modular services, which communicate with each other through well-defined interfaces. These services are designed to perform specific business functions and can be combined to create complex applications.
In an SOA, services are autonomous and independent entities that encapsulate a specific business capability or functionality. Each service is self-contained, meaning it has its own logic,

data, and resources. Services communicate with each other using standardized protocols and data formats, such as XML or JSON, making it easier to integrate disparate systems and technologies.
One of the key benefits of SOA is the reusability of services. By designing services as modular components, they can be leveraged across multiple applications and systems, reducing development time and effort. Services can be composed and orchestrated to create new applications or adapt existing ones, promoting agility and flexibility.
SOA also enables interoperability, allowing services to be developed using different programming languages, platforms, and technologies. As long as services adhere to the defined interfaces and communication protocols, they can seamlessly interact with each other, regardless of the underlying implementation details.
Moreover, SOA promotes scalability and maintainability. As services are independent units, they can be scaled independently to meet varying demands. This modular approach also simplifies maintenance, as changes or updates to a specific service do not impact other services in the system.

Account Service

The AccountService class is a part of the server.services package. It provides methods for retrieving tests and getting the ID associated with a given nickname. The class has two private instance variables: studentPersistence of type StudentPersistence and testPersistence of type TestPersistence. These variables are used to interact with the database and retrieve student and test information. The constructor initializes the studentPersistence and testPersistence objects. The retrieveTests method takes a nickname as a parameter, finds the corresponding student ID using the studentPersistence object, and then retrieves the tests associated with that student from the database using the testPersistence object. It converts the retrieved test entries into a JSON string using the ObjectMapper class from the Jackson library. The getId method also takes a nickname as a parameter and uses the studentPersistence object to find the corresponding student ID. It then returns the ID as a string.

Admin Service

The AdminService class, part of the server.services package, handles various administrative tasks. It includes methods such as deleteStudent, which deletes a student and their associated tests from the database; approveStudent, which updates the status of a student to indicate approval; seeAllStudents, which retrieves all students and converts them into a JSON string; and seeRequests, which retrieves students with a status of "REQUESTED" and converts them into a JSON string. The class utilizes the StudentPersistence and TestPersistence objects to interact with the database and perform the necessary operations.

Language Service

The LanguageService class, part of the server.services package, facilitates language-related operations. It contains a questionsPersistence object to interact with the database and manages the updating of question language. The updateLang method takes an index parameter, deletes all existing questions using the questionsPersistence object, and creates a DatabaseCreator object to populate the questions table in the database based on the specified language index. This class provides functionality to update the language of questions by clearing the existing data and filling the table with questions from different language-specific text files using the DatabaseCreator class.

Login Service

The LoginService class, part of the server.services package, handles login-related operations. It utilizes the StudentPersistence object to find the corresponding student object based on the provided username. The retrieved student object is then converted into a JSON string using the ObjectMapper class, allowing it to be returned as the result of the login operation.

New Account Service

The NewAccountService class, part of the server.services package, is responsible for creating new user accounts. It utilizes the StudentPersistence object to check if the provided nickname already exists in the database. If the nickname is available and the required fields (name, surname, and password) are not empty, a new Student object is created with the provided information and inserted into the database. The class ensures data integrity by returning appropriate messages: "OK" for successful account creation, `"ERROR_NAME_USED"` if the nickname is already taken, and `"ERROR_EMPTY_FIELDS"` if any required field is missing or empty.

Test Service

The TestService class, part of the server.services package, is responsible for handling test-related operations. It interacts with the TestPersistence and QuestionsPersistence objects to perform database operations on tests and questions, respectively. The class includes methods such as insertTest for inserting a new test into the database with the provided student ID, points, and timestamp, and getQuestions for retrieving a JSON string representation of a set of questions based on their IDs. The TestService class encapsulates functionality related to test management and question retrieval, providing the necessary operations to interact with the underlying database components.

4.4.3 Client-server communication

Server side

The ‘Server‘ class represents a basic server implementation. It manages the functionality of starting and closing a server, as well as accepting client connections and handling them using separate threads.
The class has three instance variables: ‘port‘, an integer representing the port number on which the server will listen for client connections; ‘serverSocket‘, a ‘ServerSocket‘ object used to listen for incoming client connections; and ‘clientHandler‘, an instance of the ‘ClientHandler‘ class responsible for handling client requests.
The constructor ‘Server()‘ initializes the ‘port‘ variable with a default value of 2222. There is another constructor ‘Server(int port)‘ that allows setting a custom port value.
The ‘startServer()‘ method is responsible for starting the server. It creates a new ‘ServerSocket‘ object using the specified port and enters a continuous loop that accepts client connections. Each time a client connects, a new ‘Socket‘ object is created to represent the connection. A message is printed to the console indicating the successful client connection. A ‘ClientHandler‘ object is then instantiated with the ‘Socket‘ as a parameter, and a new ‘Thread‘ is created with the ‘ClientHandler‘ as the target. The thread is started to handle the client request in a separate thread, allowing the server to handle multiple client connections concurrently.
In case of an ‘IOException‘ while setting up or accepting client connections, an error message is printed to the console, and the server proceeds to the ‘finally‘ block. The ‘finally‘ block calls the ‘closeServer()‘ method to gracefully close the server.
The ‘closeServer()‘ method is responsible for closing the server. It checks if the ‘serverSocket‘ is

not null and closes it. If an ‘IOException‘ occurs during the closing process, an error message is printed to the console.

Client side

The ‘Client‘ class represents a basic client implementation. It manages the functionality of connecting to a server, sending and receiving messages, and closing the client connection.
The class has several instance variables: ‘port‘, an integer representing the port number on which the client will connect to the server; ‘ipAddress‘, a string representing the IP address of the server; ‘socket‘, a ‘Socket‘ object representing the client-side socket connection; ‘bufferedWriter‘, a ‘BufferedWriter‘ object used for writing messages to the server; and ‘bufferedReader‘, a ‘BufferedReader‘ object used for reading messages from the server.
The constructor ‘Client()‘ initializes the ‘port‘ variable with a default value of 2222 and the ‘ipAddress‘ variable with the loopback address ”127.0.0.2”. There is another constructor ‘Client(int port, String ipAddress)‘ that allows setting custom port and IP address values.
The ‘connectClient()‘ method is responsible for establishing a connection to the server. It creates a new ‘Socket‘ object with the specified IP address and port. If the connection is successful, messages indicating the successful connection are printed to the console. The method also initializes the ‘bufferedWriter‘ and ‘bufferedReader‘ objects with the respective streams from the socket to enable communication with the server. If an ‘IOException‘ occurs during the connection process, the ‘closeAll()‘ method is called to gracefully close the client resources.
The ‘sendMessage(String message)‘ method sends a message to the server. It writes the message to the ‘bufferedWriter‘, followed by a newline character and flushes the writer to ensure the message is sent immediately. If an ‘IOException‘ occurs during the writing process, the ‘closeAll()‘ method is called to close the client resources.
The ‘receiveMessage()‘ method reads a message from the server. It reads a line of text from the ‘bufferedReader‘ and returns it as a string. If an ‘IOException‘ occurs during the reading process, the ‘closeAll()‘ method is called to close the client resources.
The ‘closeAll(Socket socket, BufferedReader bufferedReader, BufferedWriter bufferedWriter)‘ method is responsible for closing all client resources. It checks if each resource is not null and proceeds to close them. If an ‘IOException‘ occurs during the closing process, an error message is printed to the console.

Message handling

The ‘ClientHandler‘ class implements the ‘Runnable‘ interface and represents a handler for client connections in the server. It manages the communication with a specific client by receiving messages from the client, processing the commands, and sending appropriate responses back to the client.
The class has several instance variables, including ‘socket‘, a ‘Socket‘ object representing the client connection; ‘bufferedReader‘, a ‘BufferedReader‘ object used for reading messages from the client; and ‘bufferedWriter‘, a ‘BufferedWriter‘ object used for sending messages to the client. It also creates instances of various service classes: ‘AccountService‘, ‘AdminService‘, ‘LoginService‘, ‘NewAccountService‘, ‘TestService‘, ‘LanguageService‘, and ‘ProxyImageServer‘.
The constructor ‘ClientHandler(Socket socket)‘ initializes the instance variables and sets up the input and output streams for communication with the client.
The ‘broadcastMessage(String message)‘ method writes a message to the ‘bufferedWriter‘ and flushes it to send the message to the client. If an ‘IOException‘ occurs during the writing process, the ‘closeAll()‘ method is called to close the client resources.
The ‘run()‘ method is the main logic of the ‘ClientHandler‘. It runs in a loop while the socket

is connected to the client. It reads messages from the client using the ‘bufferedReader‘ and processes the received commands. Depending on the command, it invokes the corresponding methods from the service classes to perform the required operations and obtains the results. The results are then broadcasted to the client by calling the ‘broadcastMessage()‘ method.
The ‘closeAll(Socket socket, BufferedReader bufferedReader, BufferedWriter bufferedWriter)‘ method is responsible for closing all client resources. It checks if each resource is not null and proceeds to close them. If an exception occurs during the closing process, it is printed to the console.