**Tourist Reservation and Management System Using Java**

In the contemporary travel industry, efficient and user-friendly systems are essential for both tour operators and tourists. This project proposal outlines the development of a Tourist Agency System, designed using Java and JFrame for GUI development. The system aims to facilitate seamless reservations for both individual and group tourists, integrating diverse packages and payment strategies. This proposal outlines the development of an advanced Tourist Agency System, designed using Java and JFrame for GUI development.

**Objectives**:

* To develop a system for managing tourist reservations.
* To provide diverse package options for group reservations, including transportation, accommodation, and activities.
* To implement a secure and versatile payment system accommodating multiple payment strategies.
* To utilize design patterns for efficient and scalable system architecture.

**System Features:**

* Reservation Module: Individual and group reservation capabilities.
* Group packages (e.g., bus transport, hotel bookings, sea diving activities).
* Dynamic Package Customization: Builder Pattern for creating tailored travel packages.
* Payment Module: Implementation of Strategy Pattern for various payment methods (credit card, PayPal, etc.).
* Secure transaction processing.
* Process Tracking and Recovery: Memento Pattern for tracking reservation processes and undoing actions if necessary.
* Chat with Support: The Observer Pattern is used to handle chat messages between users and customer support.
* Dynamic Individual Offer Adding: Decorator Pattern is used to add luxury features to selected individual offers (such as Luxury Taxi etc.)

**Design Patterns**

**Strategy Pattern for Payment Methods**

The Strategy Pattern is used to implement different payment methods (PayPal and Credit Card) in a flexible and interchangeable way. The Strategy Pattern allows us to define a family of algorithms, encapsulate each one, and make them interchangeable.

* **PaymentStrategy Interface:** The PaymentStrategy interface defines a contract for payment strategies. It declares a single method pay(int amount) for processing payments.
* **PayPalStrategy Class:** The PayPalStrategy class implements the PaymentStrategy interface. It has instance variables email and password to store the user's PayPal email and password. The pay(int amount) method in this class prints a message to the console and displays a message box (using JOptionPane) indicating that the payment was successful with PayPal. The toString() method is overridden to provide details about the PayPal payment method.
* **CreditCardStrategy Class:** The CreditCardStrategy class also implements the PaymentStrategy interface. It has instance variables name, cardNumber, cvv, and dateOfExpiry to store credit card details. The pay(int amount) method in this class prints a message to the console and displays a message box (using JOptionPane) indicating that the payment was successful with a credit card.

**Builder Pattern for Customizable Packages**

The Builder Pattern is a creational design pattern that separates the construction of a complex object from its representation, allowing us to create different variations of an object step by step.

* **Package Class**: The Package class represents a package that can include transportation (t), accommodation (a), and activities (act). It provides methods to set these components and calculate the total price of the package (getTotalPricee()).
* **PackageBuilder Abstract Class**: The PackageBuilder abstract class serves as the builder for creating packages. It contains an instance variable p of type Package, which is the package being constructed. The createPackage() method initializes a new Package object and starts the construction process. The abstract methods buildTransportation(), buildAccommodation(), and buildActivities() are declared. Subclasses will implement these methods to specify the details of each component of the package. The getP() method allows access to the constructed package. The setP() method allows setting an existing package if needed (though it's not used in this example).
* **Offer Class (Concrete Builder):** The Offer class is a concrete subclass of PackageBuilder. It provides a specific implementation for building a package of type Offer. The createPackage() method initializes a new Package object and specifies the components for an Offer package using the buildTransportation(), buildAccommodation(), and buildActivities() methods.

**Memento Pattern for Process Tracking**

The Memento Pattern is a behavioral design pattern used for capturing and storing the current state of an object in a manner that it can be restored to that state later, without breaking the encapsulation of the object.

* **CareTaker Class**: The CareTaker class is responsible for managing and saving mementos (in this case, reservations) of customers and their services. It uses a TreeMap called newMaptoSave to store reservations with phone numbers as keys. When a reservation needs to be saved, it's added to the newMaptoSave. The saveMemento method takes a phone number and a Reservation object, adds the reservation to the map, and then serializes the entire map to a file using Java Object Serialization.
* **Customer Class**: The Customer class represents customer information and preferences. It implements the Serializable interface, indicating that its instances can be serialized and deserialized. Customer information includes name, phone, age, job, and various preferences (e.g., luxury hotel, luxury taxi, adventure sea cruise).
* **Originator Class**: The Originator class creates and saves reservations. It takes a Customer object, a list of Services, a CareTaker object, and a PaymentStrategy object as parameters during initialization. The createReservation method creates a new Reservation object using the provided customer, services, and payment strategy. It then asks the CareTaker to save this reservation, effectively capturing the current state.
* **Reservation Class**: The Reservation class represents a reservation, which includes a customer, a list of services, and a payment strategy. Like the Customer class, it implements the Serializable interface.

**The Observer Pattern for Chat with Support**

The observer pattern is a behavioral design pattern used to establish a one-to-many dependency relationship between objects. It allows one object (the subject) to notify multiple dependent objects (observers) about changes in its state or data.

The ChatSubject acts as the subject, while ChatObserver objects represent the observers. When a message is sent, the notifyObservers method is called to inform all relevant observers about the message, allowing them to react accordingly. This decouples the sender (sender of chat messages) from the receiver (chat observers) and enables efficient communication between different parts of the application.

**The Decorator Pattern for Dynamically Adding Luxury Features**

The Decorator Pattern is a structural design pattern, which is used to add new functionality to an object dynamically without altering its structure. The Decorator Pattern is used here to enhance services such as accommodations and transportation with additional features and prices.

* **Services Class Hierarchy**: There's an abstract base class called Services which represents various services offered by a tourist agency. Accommodations and Activities are abstract subclasses of Services, representing specific types of services.
* **ServiceDecorator Class**: ServiceDecorator is an abstract subclass of Services that acts as a decorator base class. It contains an instance variable decoratedService, which holds a reference to the service being decorated.
* **Concrete Decorators**: There are concrete decorator classes like LuxuryHotel and AdventureSeaCruise that extend ServiceDecorator. These decorators take a Services object (the service to be decorated) in their constructor.

**UML Class Diagram**

