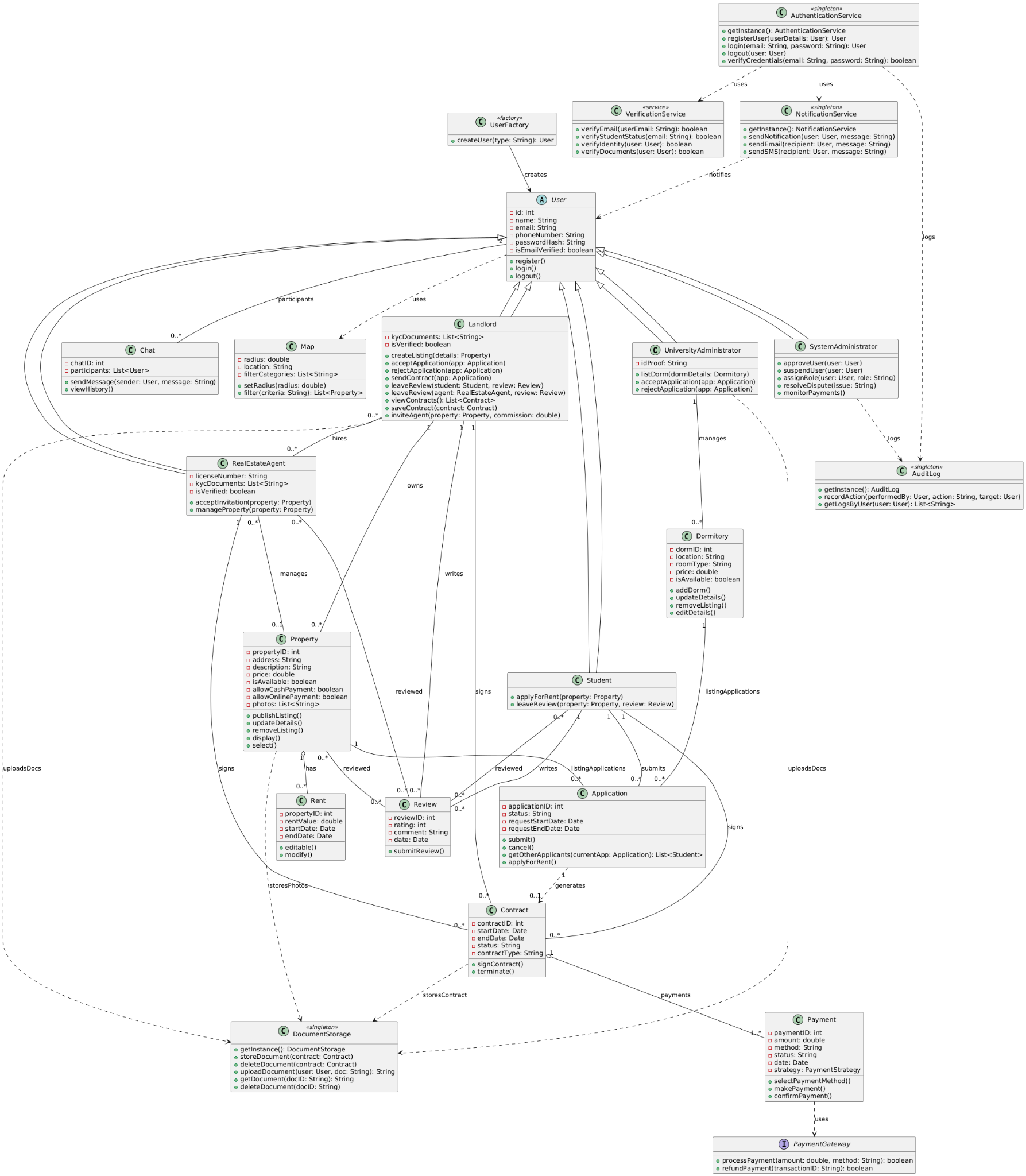
# 6. Design Patterns

1. **Factory Pattern**

* Updated Class Diagram with the Factory Pattern



* Explanation

The Factory Pattern is used in this system to manage the creation of different user types such as Student, Landlord, RealEstateAgent, UniversityAdministrator, and SystemAdministrator. Each of these roles inherits from the abstract User class, but depending on the context—such as a user registering or logging in—the system must determine which subclass to instantiate. Rather than relying on complex if-else or switch logic scattered throughout the codebase, a dedicated class called UserFactory is introduced. This factory encapsulates the instantiation logic and exposes a single method, such as createUser(String type), that returns the correct User subclass based on the input type. This improves scalability and maintainability by centralizing the creation logic, ensuring the rest of the system remains decoupled from specific subclass instantiation. It also adheres to the Open/Closed Principle, allowing new user roles to be introduced with minimal changes to existing code. By using the Factory Pattern, the system becomes cleaner, easier to extend, and more robust in handling future requirements related to user creation.

Without the Factory Pattern, we would need to instantiate users like this:

User u;

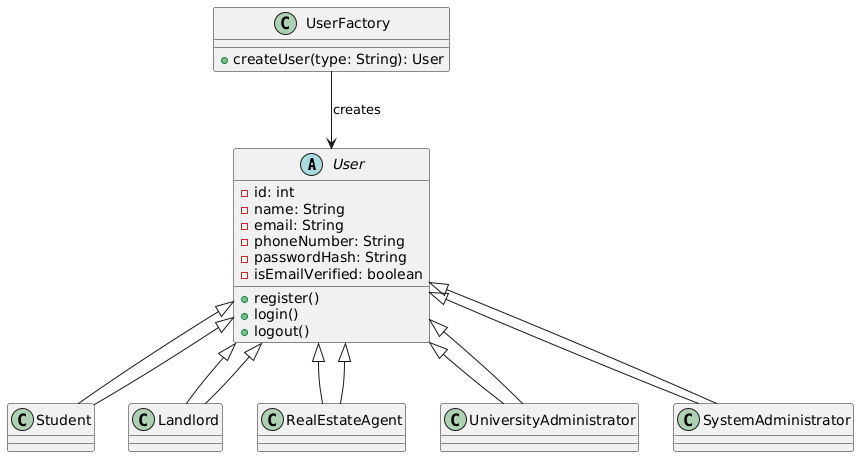
if (type.equals("student")) u = new Student();

else if (type.equals("landlord")) u = new Landlord();

// ...and so on

That’s why using the factory pattern is a good approach.

* Factory Pattern in our system



* Short Java Snippet

// Abstract base class

abstract class User {

String name;

String email;

// other fields and methods

}

// Subclasses

class Student extends User {}

class Landlord extends User {}

class RealEstateAgent extends User {}

class UniversityAdministrator extends User {}

class SystemAdministrator extends User {}

// Factory class

class UserFactory {

public static User createUser(String type) {

switch (type.toLowerCase()) {

case "student":

return new Student();

case "landlord":

return new Landlord();

case "agent":

return new RealEstateAgent();

case "universityadmin":

return new UniversityAdministrator();

case "systemadmin":

return new SystemAdministrator();

default:

throw new IllegalArgumentException("Invalid user type: " + type);

}

}

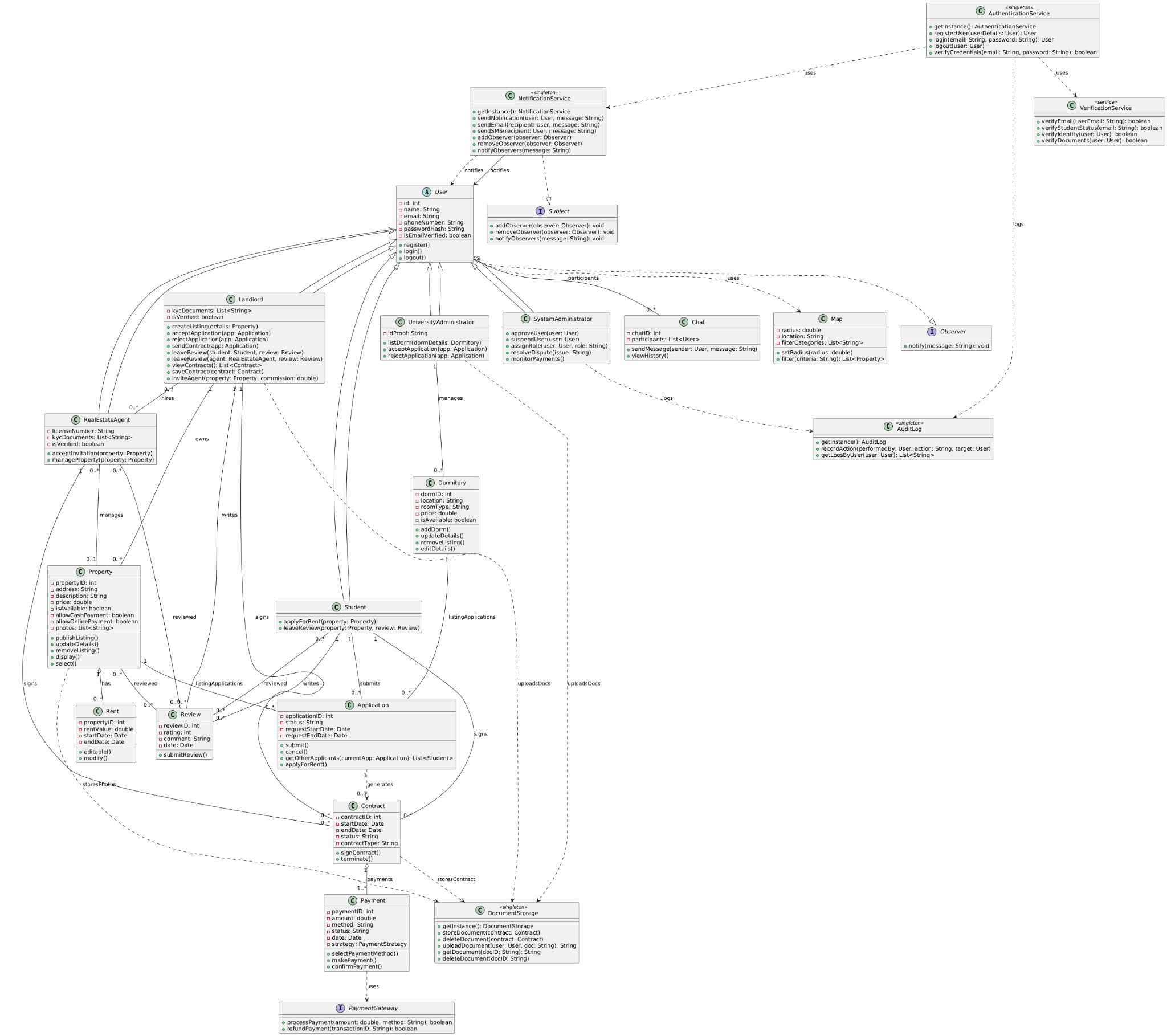
}

Usage:  
User user1 = UserFactory.createUser("student");

User user2 = UserFactory.createUser("landlord");

1. **Observer Pattern**

* Updated Class Diagram with the Observer Pattern



* Explanation

The Observer Pattern is used in our system to enable real-time, event-driven communication between the NotificationService and different user roles (such as Student, Landlord, and RealEstateAgent). This pattern establishes a one-to-many dependency between a subject (also called the publisher) and observers (subscribers), so that when the subject's state changes, all registered observers are automatically notified. In our design, the NotificationService acts as the Subject. It maintains a list of registered observers and notifies them whenever a message or event occurs (e.g., contract approval, application updates, property changes). All subclasses of User act as Observers, meaning they can register with the NotificationService and be notified when relevant events occur. This pattern is highly useful in systems where multiple components need to react to dynamic updates without tight coupling. For example, when a landlord posts a new listing, students who have subscribed to similar listings can be notified instantly. Or, if a student’s rental application is accepted, the system can automatically alert the student and update the dashboard.

Why We Use It

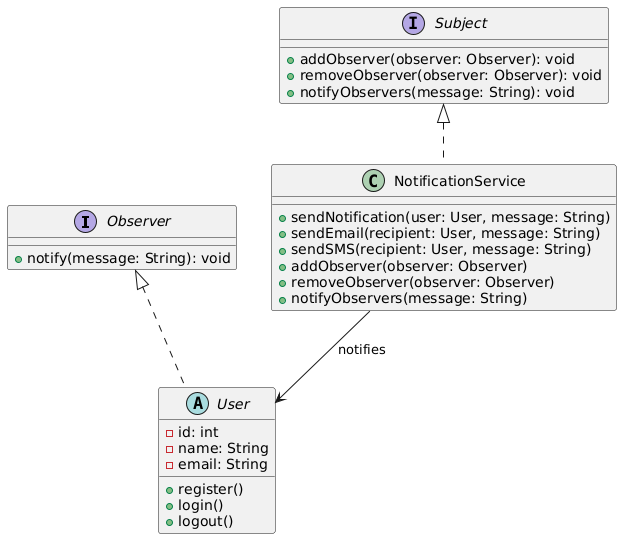
Loose coupling between NotificationService and user types — the service doesn’t need to know which concrete users are observing.

Scalability — multiple observers (students, landlords, agents) can be notified of different events without modifying core logic.

Reusability — new observer types (like a ParentObserver or FinancialAdvisorObserver) can be added easily.

Real-world realism — the system mirrors event-driven apps like rental platforms, job boards, or notification feeds.

* Observer Pattern in our system



* Short Java Snippet

// Step 1: Observer interface

interface Observer {

void notify(String message);

}

// Step 2: Subject interface

interface Subject {

void addObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers(String message);

}

// Step 3: Concrete Observer (e.g., User)

class User implements Observer {

String name;

public User(String name) {

this.name = name;

}

public void notify(String message) {

System.out.println(name + " received: " + message);

}

}

// Step 4: Concrete Subject (NotificationService)

class NotificationService implements Subject {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers(String message) {

for (Observer o : observers) {

o.notify(message);

}

}

}

//Example Usage

public class Main {

public static void main(String[] args) {

NotificationService service = new NotificationService();

User student = new User("Alice");

User landlord = new User("Bob");

service.addObserver(student);

service.addObserver(landlord);

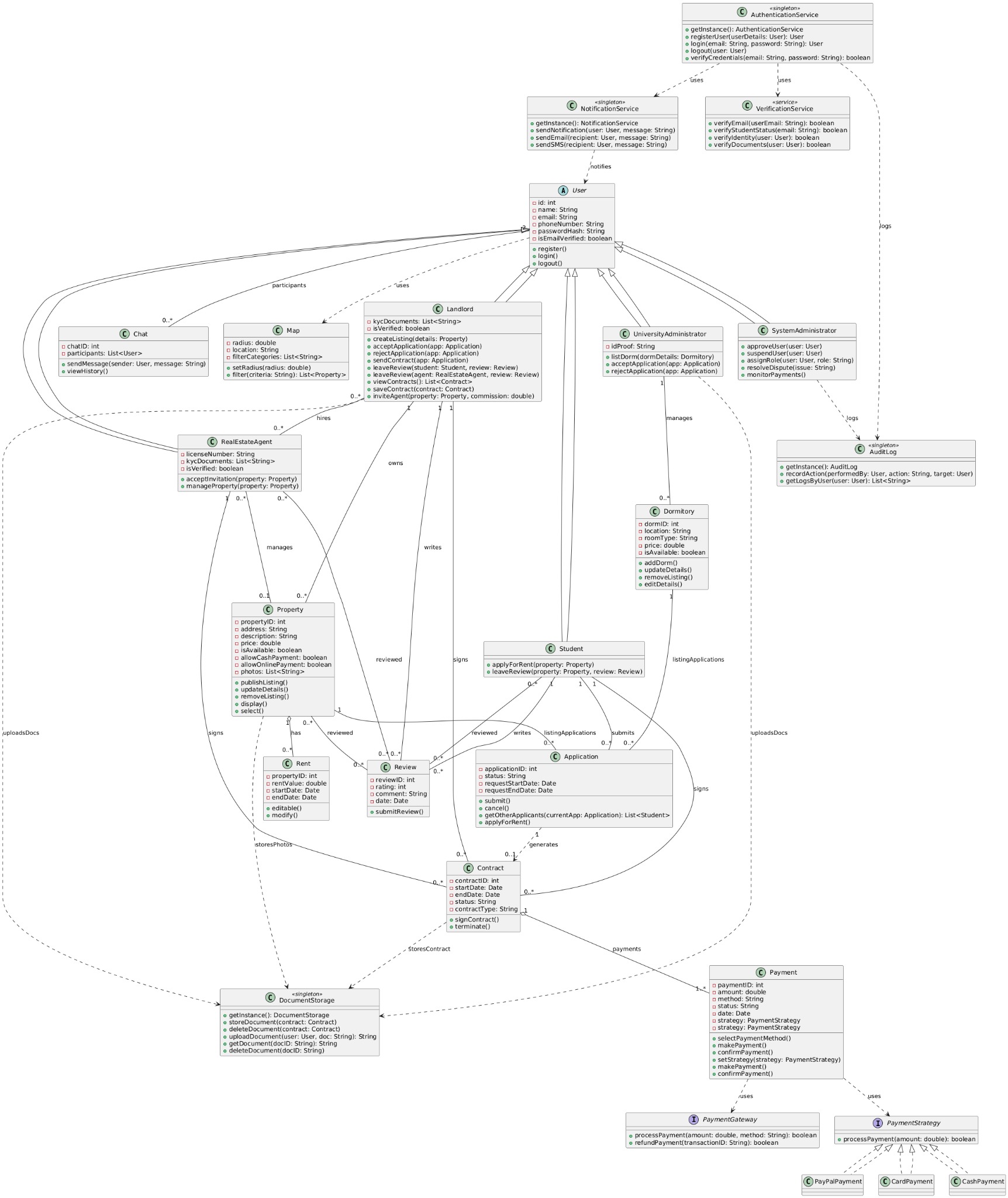
service.notifyObservers("A new property listing is available.");

}

}

1. **Strategy Pattern**

* Updated Class Diagram with the Strategy Pattern



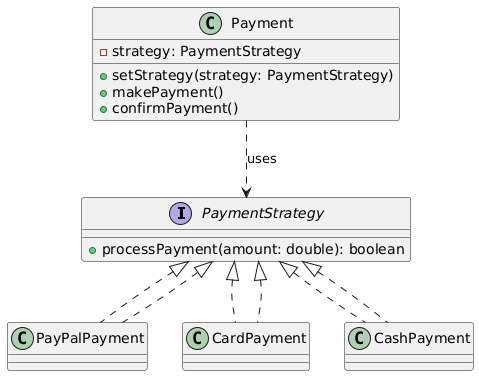
* Explanation

The Strategy Pattern is used in our system to provide a flexible and scalable way of handling different payment methods such as PayPal, Card, and Cash. Instead of embedding conditional logic within the Payment class to handle each type, we delegate the behavior to separate strategy classes that implement a common interface called PaymentStrategy. This enables us to encapsulate each payment method’s logic independently, while still adhering to a consistent contract.

In our implementation, we define a PaymentStrategy interface with a processPayment() method. Each concrete strategy class—PayPalPayment, CardPayment, and CashPayment—implements this method with its own logic. The Payment class acts as the context: it holds a reference to a PaymentStrategy and delegates the actual payment processing to it. This way, we can change the strategy at runtime simply by calling setStrategy() with the desired payment method.

The primary advantage of using the Strategy Pattern in our system is that it promotes loose coupling between the Payment class and the specific payment mechanisms. It also follows the Open/Closed Principle, allowing us to add new payment methods in the future without modifying the existing Payment class. This structure also simplifies testing, as each strategy can be verified independently. Ultimately, the Strategy Pattern provides a clean, modular approach to managing varied but interchangeable behaviors in a system like ours where payment flexibility is essential.

* Strategy Pattern in our system



* Short Java Snippet

// Step 1: Define the strategy interface

interface PaymentStrategy {

void processPayment(double amount);

}

// Step 2: Implement concrete strategies

class PayPalPayment implements PaymentStrategy {

public void processPayment(double amount) {

System.out.println("Paid $" + amount + " via PayPal.");

}

}

class CardPayment implements PaymentStrategy {

public void processPayment(double amount) {

System.out.println("Paid $" + amount + " via Card.");

}

}

class CashPayment implements PaymentStrategy {

public void processPayment(double amount) {

System.out.println("Paid $" + amount + " in Cash.");

}

}

// Step 3: Context class that uses a strategy

class Payment {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

strategy.processPayment(amount);

}

}

//Example Usage

public class Main {

public static void main(String[] args) {

Payment payment = new Payment();

payment.setStrategy(new PayPalPayment());

payment.makePayment(50.0); // Output: Paid $50.0 via PayPal.

payment.setStrategy(new CardPayment());

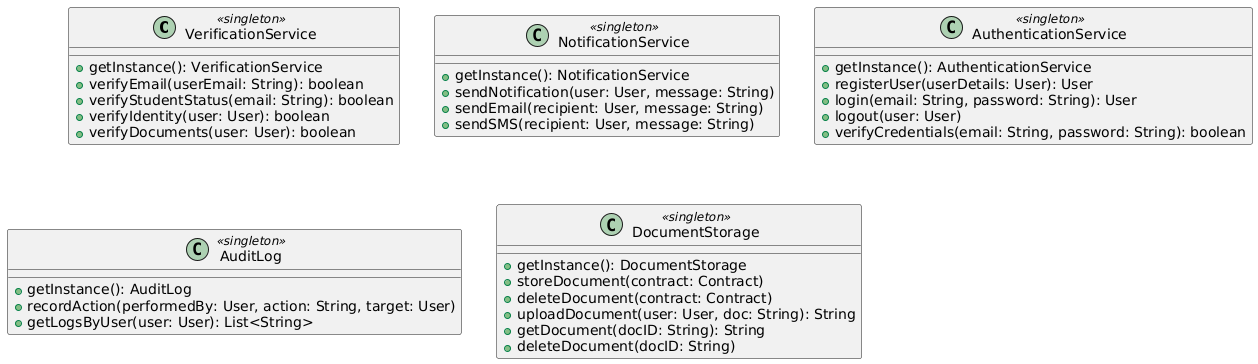
payment.makePayment(75.0); // Output: Paid $75.0 via Card.

}

}

1. **Singleton Pattern**

* Singleton Pattern in our system



* Explanation

The Singleton Pattern is used in our system to ensure that certain core services are instantiated only once and accessed globally throughout the application. These include NotificationService, AuthenticationService, AuditLog, DocumentStorage, and now also VerificationService. Each of these classes performs centralized and shared functionality, such as sending notifications, authenticating users, logging system activity, verifying identities, or managing documents. By making them singletons, we ensure consistent behavior, reduce memory usage, and prevent conflicts caused by multiple instances.

In our implementation, each singleton class provides a static method called getInstance(), which returns the same instance every time it is called. The constructor of each class is made private to prevent outside code from creating new instances. This approach guarantees that there is always exactly one instance of these critical services available to the system.

We chose to use the Singleton Pattern because it simplifies the management of shared services. Instead of passing service instances throughout the system or recreating them unnecessarily, we access them via their global access point. This is especially useful for services like AuditLog, which must capture all user actions in a consistent and centralized manner, and NotificationService, which needs to broadcast messages system-wide. It also ensures uniform enforcement of security rules in AuthenticationService and reliable identity checks via VerificationService.

By applying the Singleton Pattern, we maintain tight control over global service behavior, promote resource efficiency, and ensure that shared components behave consistently across all system modules.

* Short Java Snippet

// Step 1: Singleton class example - NotificationService

public class NotificationService {

// Static instance (eager or lazy instantiation can be used)

private static NotificationService instance;

// Private constructor prevents direct instantiation

private NotificationService() {}

// Public access point for the singleton instance

public static NotificationService getInstance() {

if (instance == null) {

instance = new NotificationService();

}

return instance;

}

// Sample service method

public void sendNotification(User user, String message) {

System.out.println("Notifying " + user.getName() + ": " + message);

}

}

//Example Ussage

public class Main {

public static void main(String[] args) {

NotificationService service = NotificationService.getInstance();

User student = new User("Alice"); // Assume basic constructor for example

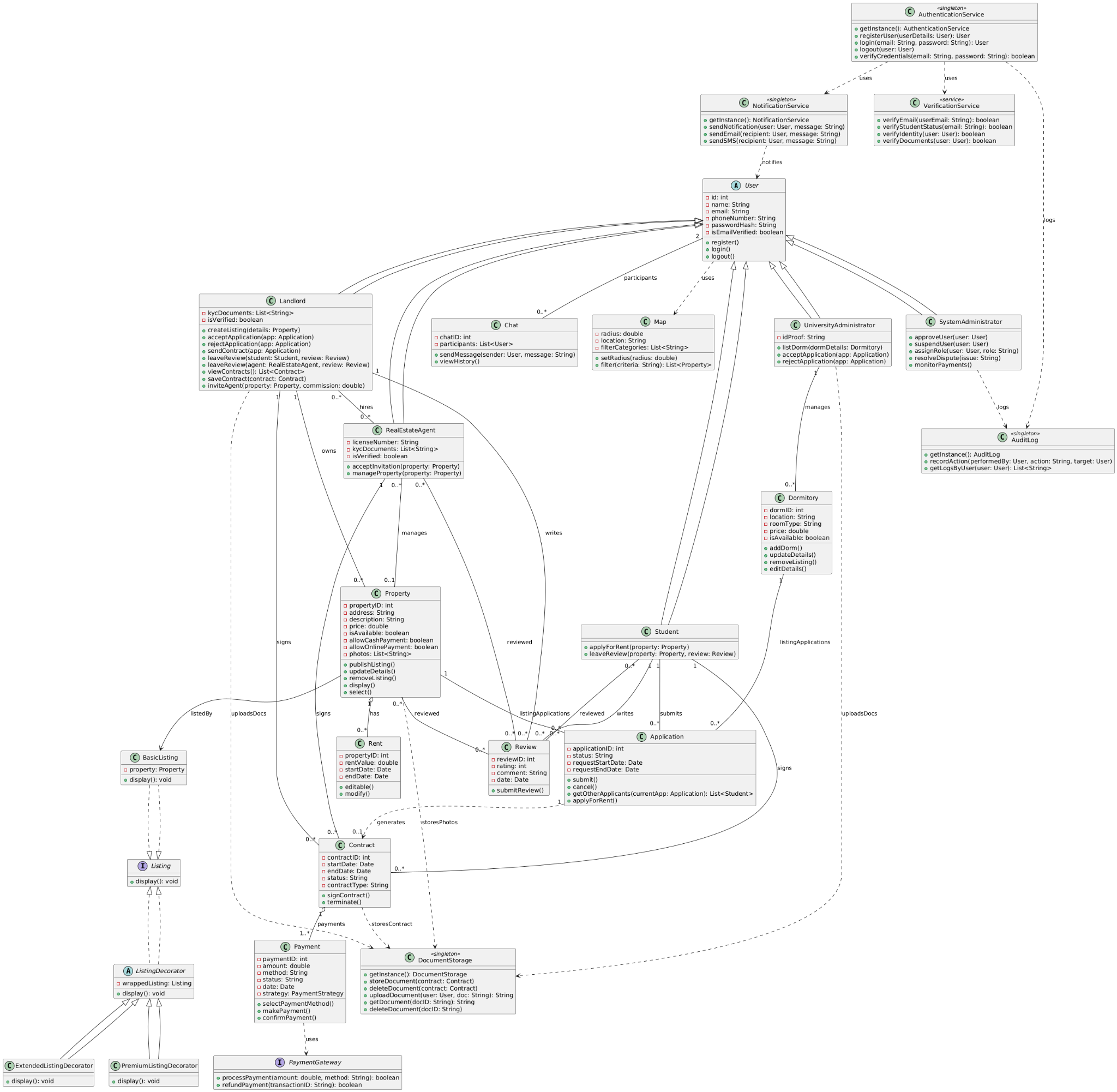
service.sendNotification(student, "Your contract has been approved.");

}

}

1. **Decorator Pattern**

* Updated Class Diagram with the Decorator Pattern



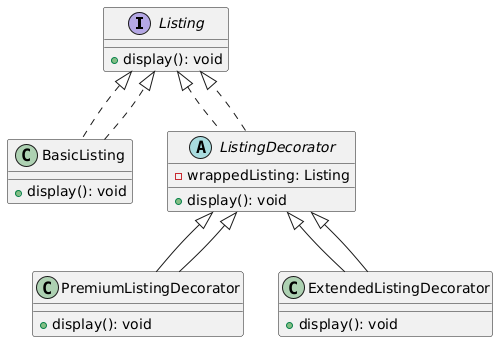
* Explanation

The Decorator Pattern is used in our system to allow dynamic enhancement of property listings without modifying the original Listing logic. In a platform like ours, different listings may require different types of display features. For example, some listings might be basic, while others may be highlighted as premium, or include extended metadata such as neighborhood insights or promotional banners. Instead of adding all these variations inside one Listing or Property class through inheritance or conditionals, we apply the Decorator Pattern to layer these features modularly and flexibly.

We start by defining a Listing interface with a display() method. The base class BasicListing implements this interface and wraps around a Property object, acting as the default representation. We then introduce an abstract class called ListingDecorator which also implements Listing and wraps another Listing instance. This makes it possible to chain decorators dynamically. Specific decorators such as PremiumListingDecorator and ExtendedListingDecorator extend ListingDecorator and add their own behavior to the display() method while delegating the base logic to the wrapped component.

This pattern is particularly powerful for our system because it gives us full flexibility to stack enhancements as needed, supports open/closed design (new decorators can be added without touching existing ones), and avoids code duplication. For instance, a landlord may choose to upgrade a property to both premium and extended formats by simply applying both decorators in sequence. As a result, the Decorator Pattern provides us with a scalable, readable, and maintainable way to handle diverse and evolving listing formats across our platform.

* Decorator Pattern in our system



* Short Java Snippet

// Step 1: Listing interface

interface Listing {

void display();

}

// Step 2: Basic listing that wraps a Property

class Property {

String address;

public Property(String address) {

this.address = address;

}

}

class BasicListing implements Listing {

private Property property;

public BasicListing(Property property) {

this.property = property;

}

public void display() {

System.out.println("Listing: " + property.address);

}

}

// Step 3: Abstract decorator

abstract class ListingDecorator implements Listing {

protected Listing wrappedListing;

public ListingDecorator(Listing listing) {

this.wrappedListing = listing;

}

public void display() {

wrappedListing.display();

}

}

// Step 4: Concrete decorators

class PremiumListingDecorator extends ListingDecorator {

public PremiumListingDecorator(Listing listing) {

super(listing);

}

public void display() {

super.display();

System.out.println("This listing is marked as PREMIUM.");

}

}

class ExtendedListingDecorator extends ListingDecorator {

public ExtendedListingDecorator(Listing listing) {

super(listing);

}

public void display() {

super.display();

System.out.println("Includes extended property details and map features.");

}

}

// Example Usage  
public class Main {

public static void main(String[] args) {

Property property = new Property("123 Maple Street");

Listing listing = new BasicListing(property);

listing = new PremiumListingDecorator(listing);

listing = new ExtendedListingDecorator(listing);

listing.display();

}

}  
  
  
//Output  
Listing: 123 Maple Street

This listing is marked as PREMIUM.

Includes extended property details and map features.