Observer Pattern

NotificationSystem

The **NotificationSystem** class embodies the **Observer Pattern** by acting as a central publisher of notifications across the property management platform. Entities like Administrator, Tenant, Landlord, and **PropertyManager** effectively act as **observers**, each capable of triggering or responding to notifications based on system events. For instance, when an Administrator suspends or deactivates a user account, or prepares an announcement, a notification is generated through **sendNotification()**, which then propagates to the intended recipients via the **NotificationSystem**.

This class maintains a Messages list, serving as a shared observable state. Methods like **timeHasArrived()** help determine whether any pending messages should be dispatched. **The system patiently waits** for a valid notification event to occur before broadcasting it, ensuring only meaningful messages are distributed. By isolating all outbound messaging logic within a single class and keeping the notification logic decoupled from domain logic (like lease management or maintenance requests), this pattern promotes loose coupling and centralized control.

The design reflects a **classic publisher-subscriber model**, where changes in system state (like a new maintenance issue or user ban) do not require all affected classes to be directly linked — instead, they rely on a shared notification channel, improving modularity and change tolerance.

FeedbackSystem

The **feedbackSystem** class applies the **Observer Pattern** by serving as a centralized hub for collecting and broadcasting user feedback in a reactive manner. It holds a **feedbackList** which is a growing stream of reviews and ratings tied to properties and provides interfaces that support dynamic observation and response to tenant activity.

When a Tenant submits a rating or comment (**rateProperty()**), the **feedbackSystem** is updated, which in turn reflects on the linked Property objects through shared references like reviews. **Before activating feedback collection, the system passively waits for a user to rent and live in the property**, ensuring that only experienced users contribute evaluations. By implementing methods such as **navigateFeedback()** and **requireFeedback()**, the system monitors user engagement and prompts actions like prompting feedback submission after property viewings or lease conclusions.

The Property class observes changes within the feedbackSystem indirectly feedback data affects how properties are evaluated or displayed. This setup reflects the Observer Pattern's structure: one or more **observers** (Property, possibly Landlord or Manager) react to changes in the **observable** (**feedbackSystem**), without being tightly bound to the internals of that system.

By decoupling data input (tenant reviews) from data consumption (property management or UI analytics), the design enables scalable and maintainable feature extensions like reputation scoring or automatic moderation triggers.