The Composite Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies.

1. This enables clients to treat individual objects and compositions of objects uniformly.  
   **Component Interface:**
   * **BankComponent:**
     + This is an interface that declares a method **getInfo()** for managing and accessing properties of components in the system. All components will implement this interface to provide their specific details.
2. **Leaf Objects:**
   * **Transaction:**
     + Represents a transaction with attributes like transaction ID and amount. It has a method **process()** to handle the transaction logic.
   * **Customer:**
     + Represents a customer with attributes like customer ID and name. Methods include **addAccount()** and **getAccounts()** to manage customer accounts.
   * **Loan:**
     + Represents a loan with attributes such as loan ID, amount, time, interest rate, and monthly payment. Methods include **calculatePayment()** to calculate monthly payments, **approveLoan()** to approve the loan, and **rejectLoan()** to reject the loan.
3. **Composite Objects:**
   * **Account:**
     + Represents a bank account with attributes such as account number and balance. Methods like **deposit()**, **withdraw()**, and **getBalance()** provide foundational functionalities.

**Usage Justification:**

1. **Component Interface (BankComponent):**
   * By having a common interface, the system ensures that all components (Account, Transaction, Customer, Loan) provide a uniform way to access their information, making the system modular and extensible.
2. **Leaf Objects (Account, Transaction, Customer, Loan):**
   * **Transaction:**
     + Handles individual transactions, encapsulating transaction-specific logic.
   * **Customer:**
     + Manages customer information and associated accounts, providing methods to add and retrieve accounts.
   * **Loan:**
     + Manages loan-specific data and logic, including loan calculations and approval processes. This encapsulates the complexity of loan management within a dedicated class.
3. **Composite Objects (Account):**
   * **Account:**
     + Represents individual bank accounts, providing essential banking functionalities like deposit and withdrawal.
4. **Uniform Treatment:**
   * The composite pattern allows for the uniform treatment of individual and composite objects. Methods that operate on **Account** objects can handle both simple and complex objects without knowing their concrete classes. This simplifies the code and enhances flexibility.

**Conclusion:**

Implementing this composite pattern in an online banking system enhances modularity, scalability, and maintainability. It allows the system to grow by adding new components without significant changes to the existing codebase, promoting easier maintenance and potential expansion of system functionalities. The addition of the **Loan** class further enriches the system by encapsulating loan-specific details and operations, making the system more comprehensive and robust.