Facade Pattern

Team members: *Bingcan Zhou (b25zhou), Jinghan Wu (j244wu), Vianna Vuong (vkvuong)*

Static Description:

The Facade Pattern is a widely used software design pattern often used in conjunction with object-oriented programming. Facade pattern wraps a single system or multiple subsystems with a unified interface. It normally has a wrapper class that contains a set of member methods required by the clients. The clients can call these members directly and the member hides the implementation details.

Dynamic description:

A complex system is usually designed with many independent classes, or sometime the source code is not accessible. Overtime, updates on the hidden systems can be easily performed without affecting the client. The facade layer then needs to be extended to reflect these changes.

Does the Pattern have a Purpose/Motivation?

1. Make the software easier to use, test and maintain; it provides convenient methods for common tasks (that potentially require calls to multiple subsystems).
2. Make the library more readable.
3. Reduce dependencies between libraries, systems or other packages.
4. Promotes loose coupling: communication between modules are facilitated through method calls.

Vocabulary:

1. Client: makes calls to the facade for particular services
2. Facade: serves the client by making calls to subsystems for specific services
3. Subsystem: implements the services called on by the facade

Facade is useful when:

- a simple interface to a/several complex subsystem(s) is needed
- there are many dependencies between clients and the implementation classes
- subsystems can and should be layered. If the subsystems are dependent, injecting a facade and making subsystems communicate through this facade, can simplify the dependencies that exist between them.

Advantages	Disadvantages
- Abstracts away the implementation details from the client. This provides the following benefits: - Easy to change	- Clients are unaware of/unable to use services that are not made available through the interface.

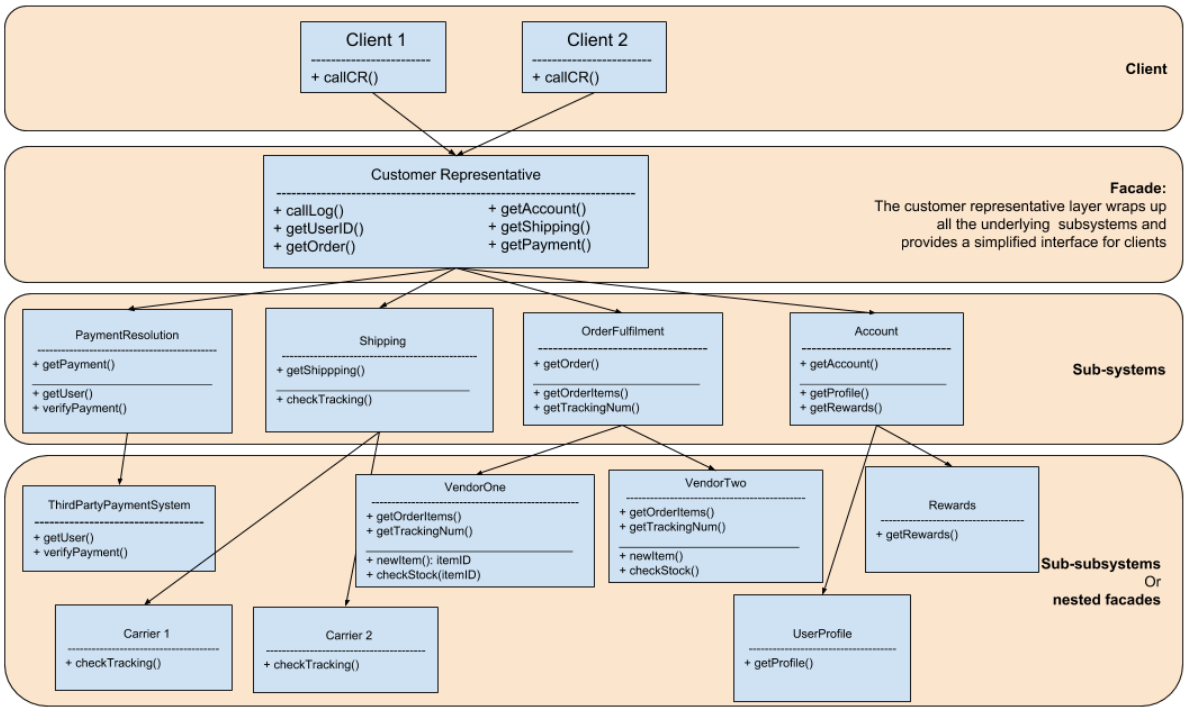
implementation of subsystems without affecting client - Simple for the client to use and understand - Potentially wrap many poorly designed APIs into a single well designed API - Decoupling of the subsystems from the clients. This will also make it easier to test and maintain.	- If the subsystems change, the facade layer must also change.

Non-functional Properties:

Improved	Degraded
<u>Scalability</u> Easily add services by adding to the facade layer. A subsystem can also be a facade itself, so it is possible to compose these systems easily. <u>Complexity/Usability/Maintainability</u> Client only needs to communicate with one system (the facade) in order to use services on multiple systems. This makes more user friendly for the client; requires less learning time because client is only "learning" about the facade. <u>Evolve-ability</u> Subsystems can be implement without affecting the client.	<u>Reliability</u> Since there is usually only one facade object, there exists a potential single point of failure. <u>Evolve-ability:</u> If you remove a subsystem, facade has to change

<u>Structure or Runtime Behaviour Example</u>: *(please see diagram below)*
By using properties from the facade pattern, clients are not required to have knowledge about the systems beyond what is presented to them through the facade object. Customer representatives acts as a simple user interface for clients to communicate with the subsystems such as shipping, order fulfilment, payments and accounts and rewards. For larger systems, nested facades may be used. In our example, customer representative is the facade. Subsystems like order fulfilment and shipping have their own subsystems; order fulfilment could be provided by different vendors and shipping could be carried out by various carriers. Additional carriers/vendors could be added with minimal effect to the clients (although the customer representatives will have to train for those new subsystems). If the customer service layer is "down", the client is unable to reach out directly to the subsystems (e.g., vendors) for

service. Clients are also unable to use subsystem services not made available through the facade. In the example below, if the carrier had a service that allowed the user to receive a text message when the parcel arrived, the client would not be able to use this feature unless the facade serviced it.



Please note methods used in the diagram are not complete, these functions are for demonstration purposes only.

Detailed Example Cases:
Our example is about online shopping. In this example, the client is the shopper/customer, the facade is the customer representative, the subsystems are all of the services needed to assist a customer's inquiries about their account/order.

Case 1: [Complexity/Usability] Demonstrate the base case (Facade is a unified interface to multiple systems.)
Subsystems: PaymentResolution, Shipping, OrderFulfillment and Account

- Example 1: A customer calls into customer service about her order and she wants to know if her order has been shipped. The customer representative then proceeds to pull in and aggregate information from the vendor (OrderFulfillment) and the carrier (Shipping).

- Example 2: In another situation, a customer calls in because her order was not shipped yet. The customer representative finds out that there was an issue with the payment. The billing address has a mistake so the order was not proceed. The customer then provided the correct billing address to the customer representative. The representative corrected the mistake in Account and then initiate the order in OrderFulfillment.

Case 2.1: [Scalability] Demonstrates how to add a subsystem.
New subsystem: Rewards

- Example: The customer service department decided to add a service. Now, customer service has support for it's new reward system. Customers will get an update on the new subsystem and train the customer representative for the new rewards system. Customer calls and to check her points. The customer representative gives her current point balance. The representative still supports the services before.

Case 2.2: [Scalability] Demonstrate how subsystems can be facades themselves (nested facades).
Subsystems: Shipping and Carriers

- Example: The Shipping Class is a facade for all of the carrier services. When the representative looks up an order from subsystem Shipping. The subsystem may checks across multiple carriers if the order is fulfilled by different vendors and thus different shipping methods. This information is then aggregated and returned to the customer.

Case 3: [Evolve-ability] Demonstrate what happens when a subsystem changes.
Subsystems: Vendor and Carrier

- Example: The shipping department decided that Carrier A is too expensive for their operational costs so they want to switch to Carrier B. This can be changes without affecting the client.

Case 4: [Reliability] Demonstrate the potential single point of failure.

- Example: Customer calls in and no representative answers because customer service hours are over. The customer is not able to make any inquiries.

Case 5: [Disadvantage] Demonstrate how the client cannot access services not exposed by the facade.
Subsystem: Carrier

- Example: Carrier B offers the ability to send text messages so that the receiver of the package will know when packages are delivered or if there is an exception in delivery. If customer service does not implement the ability to enable this for the customer, the customer will not be able to use this feature (enable could have many meanings, for example, providing the phone number of the client to the carrier). Since customer service does not support this feature at all in this example, the customer will not be able to use it.