External APIs Design Pattern

One of the non-functional requirements of the system is the need to integrate third-party services (such as the Google Maps API and payment gateways) while maintaining a failure rate of no more than 0.1%. This requirement is directly related to the system's reliability, resilience, and decoupling from potential failures in external services.

To satisfy this non-functional requirement, three core software design patterns were applied: Facade, Adapter, and Proxy. The selection of these patterns was intentional, as each addresses a different aspect of the problem: the Facade simplifies access to external services, the Adapter ensures standardized integration of diverse APIs within the system, and the Proxy provides reliability mechanisms such as retries, timeout handling, and fallback logic.

Specifically, the Facade Pattern (implemented via the APIIntegrationFacade class) provides a single, simplified access point to services such as placing map markers (placeMarker(event)) and completing payments (processPayment(amount)). This way, the GUI classes (e.g., MapGUI and PaymentGUI) do not need to be aware of the underlying APIs. In the UML diagram, the Facade is placed centrally, with incoming arrows from the GUI classes.

The Adapter Pattern was used in the GoogleMapsAdapter and PaymentGatewayAdapter classes, in order to translate system requests into the format required by each external service. In other words, the Adapter isolates the application from changes in the implementation details of the APIs. For example, GoogleMapsAdapter may internally convert an Event object into a format compatible with Google Maps API requests. In the diagram, Adapters are positioned directly below the Facade and are directly associated with it.

The Proxy Pattern was applied through the GoogleMapsProxy and PaymentProxy classes, which are interposed between the Adapters and the actual APIs. These Proxies introduce reliability mechanisms, such as multiple retry attempts when the first call fails (retryRequest()), exception handling for timeouts (handleTimeout()), fallback scenarios, and error logging (logFailure()). As a result, even if there is a temporary failure in the Google Maps API or the payment gateway, the system can retry or fail gracefully without disrupting the user experience. In the UML diagram, the Proxy classes are shown directly beneath the Adapters.

To make the functional role of each pattern even clearer, both the Adapters and Proxies—as well as the Facade—declare not only their primary public methods (placeMarker, processPayment) but also key internal (private) helper methods, such as mapEventToGoogleFormat(), preparePaymentRequest(), retryRequest(), and handleServiceError(). The presence of these helper methods demonstrates that the design patterns have been implemented with meaningful internal logic—not just superficially.

In summary, the final UML diagram that was designed includes:

- The GUI classes (MapGUI, PaymentGUI)

- The central Facade class (APIIntegrationFacade)
- The Adapter classes (GoogleMapsAdapter, PaymentGatewayAdapter)
- The Proxy classes (GoogleMapsProxy, PaymentProxy)
- The external APIs (GoogleMapsAPI, PaymentAPI)

All of the above are shown with clear associations, appropriate stereotypes (<>, <>, <>), and a separation of public vs. private methods.

By applying this architectural approach, the system ensures a high level of reliability, maintainability, and extensibility—exactly as required by the original non-functional specification.