Design Pattern Evaluation on A RESTful API Wrapper: A Case Study of Software Integration with An Internet Payment Gateway using Model-Driven Architecture

Lindung Pariningotan Manik¹,², Hani Febri Mustika¹, Arida Ferti Syafiandini¹
¹ Research Center for Informatics, Indonesian Institute of Sciences, Bandung, Indonesia
² STMIK Nusa Mandiri, Jakarta, Indonesia
{lindung.manik, hanifmustika, afsyafiandini}@gmail.com

Received 13 May 2019; accepted 29 November 2019

Abstract. A proper use of design patterns has proven to be very useful in the development of robust applications over time. In this paper, the design patterns are introduced in the early stage of the software development where model-driven architecture is used as the engineering approach. A RESTful internet payment gateway API (Application Programming Interface) wrapper is selected as the case study. At the beginning, Platform Independent Model (PIM) is created as the domain model. After that, the PIM is transformed into the Platform Specific Model (PSM). Before converting the PSM into the source code, three design patterns such as builder, observer, and factory pattern are added into the model. To evaluate the impacts of implementation, static analysis is used to examine the generated code before and after adding the design patterns. The result shows that the design decision increases cohesion, complexity, coupling, inheritance, and size metrics of the source codes.

1 Introduction

Nowadays, digitization of the business processes makes internet applications and technologies connected through the cloud growing enormously. Industry 4.0 era pushes stakeholders including governments, manufacturers, and business owners to integrate their applications within the organization, or even with the external organizations. A REST (Representational State Transfer) API (Application Programming Interface) is one of the ways to make these online integrations happen in the cloud. These APIs allow two or more software programs to communicate between each other.

A RESTful API is an API based on the REST technology which is often used in a web services development. It uses HTTP defined by RFC 2616 protocol to GET, PUT, POST, and DELETE data. Since HTTP calls are stateless, REST is very useful in cloud applications. REST also leverages less bandwidth, making it more suitable for internet usage. The API describes the proper way for a developer to write a software requesting services or receiving responses from other applications.

An API wrapper is a software package that contains clean functions to access the API. This wrapper could be written in any programming languages. It takes time to implement the API in a language, to figure out the right way to access the REST endpoints and to retrieve the response data. Once the API wrapper is written, it could
be shared among other developers to ease their pain when they must implement the same API in the same programming language.

In this paper, model-driven architecture is used as the software design approach for the development of the API wrapper. One of internet payment gateway API is selected as the case study. The main research goal is to evaluate the design patterns that could be used in the early stage of the API wrapper development. Design pattern is a best practice solution to a commonly occurring problem in software engineering [1]. Since Gang of Four standardized the design patterns for the first time [2], more and more object-oriented software apply them until now. Design patterns have some benefits on software maintainability [3], expandability [4], and modularity [5]. In this research, five software metrics, such as cohesion, complexity, coupling, inheritance, and size metric are evaluated to analyze the impacts of the design decisions.

2 State-of-the-art

Model Driven Architecture (MDA) developed by the Object Management Group (OMG) is a framework for software development using a system modeling language which is defined by [6]. MDA aims to enhance portability by separating business and application logic from underlying platform technology. The primary components of MDA technologies are the Platform Independent Model (PIM) which describes the structure or function of a system and the Platform Specific Model (PSM) which defines specific implementation.

Software development in the MDA starts with a PIM of an application’s or integrated system’s business functionality and behavior, constructed using a modeling language based on OMG’s MetaObject Facility (MOF), Unified Modeling Language (UML), or Common Warehouse Metamodel (CWM). This model remains stable as the technology evolves. MDA has the capability to define templates that map transformations from a PIM to a PSM and then to a working implementation on any platform, including Web Services, .NET, CORBA, Java, and others.

![MDA Transformation Diagram]

Fig. 1 Model-driven architecture transformation [7]

MDA facilitates the development of a system in abstraction and simplifies implementation of that system across a variety of platforms. The MDA has been successfully implemented in various business domain such as accounting system [8], embedded system [9], cancer research [10], human language technologies [11], also more recently enterprise e-health system [12] and Zakat calculation [13]. MDA may provide 10-20% efficiency improvement of existing systems engineering efforts once the methodology is understood and successfully adapted [14].

However, MDA could not apply the design patterns automatically either when transforming the PIM to the PSM or when converting the PSM into the code. Software designer should add it manually according to the needs since a design pattern is not a finished design that can be transformed directly into the code. It is a template of solving...
a problem that can be used in many different situations. Twenty three design patterns that are specified in [2] can be categorized into three main types. They are creational, behavioral, and structural pattern. Despite of their advantages, the abuse, misuse, or overuse of the design patterns could lead into creating bigger problems rather than solving the actual problems.

3 Case Study

In this paper, software development of integrating a mobile commerce application with an internet payment gateway provider is selected as the case study. The actor that uses the payment gateway service in the form of API is called as the merchant. It needs the service since the payment gateway facilitates its customer various payment methods such as credit card, online bank transfer, or e-money. By using the services, the merchants could increase their transaction number since their customer has various options while paying the transaction order. The merchants also can minimize the burdens since they do not need to implement the payment methods one by one.

An internet payment gateway is an instant payment service where the payment occurs in the provider webpage portal. With this service, the customer of the merchant will be redirected to a provider webpage upon checkout to complete the payment by selecting available payment methods. Besides creating order, the payment gateway service provider also serves other features for the merchants such as:
1. Query order status request, to check the status of a payment transaction.
2. Refund request, to refund a settled transaction.
3. Cancel request, to cancel a transaction regardless of its settlement status.
4. Query refund status request, to check the status of an order refund.

![Use-case diagram of the case study](image)

In terms of the security, merchant and payment gateway provider normally use cryptographic keys (private and public key pair) in agreed format when exchanging messages. As mentioned before, it would be very helpful if an API wrapper package could be shared to the merchant application developers in order to minimize the
integration efforts. The merchants could use the package as the library in their software when they need to connect the applications with the same payment gateway provider and must implement the API in the same programming language.

4 Design

This chapter explains the design of the PIM and the PSM that accommodates the requirements in the case study. Implementation of the design patterns are also discussed accordingly.

4.1 Platform Independent Model

In the domain of the API wrapper, several entities are created in the model such as BusinessApi, BusinessSpi, RestApiClient, and IpgConfig. The BusinessApi refers to interfaces or services hosted or implemented by the payment gateway, in which merchant act as the client (API caller) and payment gateway is the service provider. In this case, the BusinessApi uses a RestApiClient to post a request and to get a response. Meanwhile, BusinessSpi (service provider interface) refers to interfaces that are intended to be implemented or extended by the merchant. For example, the callback services implemented by merchant client, for which payment gateway provider is the caller and the merchant acts as the service provider. The client and the service provider exchange request and response messages in certain format (either XML or JSON) during the API or the SPI call.

![Fig. 3 Differentiation between API and SPI](image)

As a concrete example, a merchant needs to request creating order API when the customer wants to use the payment service and the payment gateway provider will give transaction identifier response in the return. After the customer finish the payment process (either using credit card, online bank transfer, or e-money) in the payment gateway provider webpage portal, the payment gateway provider call the order finished SPI to give the merchant a transaction status either success or failed. The same occurs with the refund order service since it is also an asynchronous flow.

Merchant has key attributes that are associated with the payment gateway system such as the client identifier and the private key. It’s application usually also has other technical configurations such as timeout connection time, connection pool size, and environment type. The IpgConfig entity holds these configuration attributes. Typically,
there are two types of environment when merchants start integrating their applications with a payment gateway application. The payment gateway providers normally serve sandbox and production environment. Sandbox is safe environment that mimics the real setting, where merchants can develop and test the features of their applications before its released to go live to the public. Thus, EnvironmentType enumeration is created to accommodate this situation. This entity holds some final or unchangeable properties (constants) about the payment gateway information such as the Uniform Resource Location (URL) and the public key.

Fig. 4 Platform independent model

### 4.2 Platform Specific Model

When transforming the PIM in Fig. 4 to the PSM in Fig. 5, the MDA tool adds some specific implementations in the selected platform. In this case study, Java programming language is selected as the specific implementation platform. As can be seen, the notable transformation rule is the properties encapsulation. The attributes’ modifiers in the PIM are converted from public to private by adding the getter’s and the setter’s methods. Besides that, the constructor methods are also added in each class. At this point, the Java codes could be generated automatically from the PSM by using the MDA tool.
4.3 Design Patterns

There are rooms for improvement before converting the PSM into the codes. For instance, design patterns could be introduced in the PSM in order to refine the software design. However, as stated before, design patterns must be implemented manually since MDA tool cannot recognize what patterns should be applied for such cases. In this part, the implementations of several design patterns such as builder, observer, and factory pattern are discussed.

4.3.1 Builder Design Pattern

The builder pattern, which is a creational design pattern, separates the construction of a complex object from its representation. In this case study, the builder pattern could
be implemented in creating the IpgConfig object. It is very useful when the object contains a lot of attributes. In case of creating the IpgConfig object, the mandatory parameters are a client identifier, an environment type, and a private key. Meanwhile, the optional parameters are a connected timeout, a maximum connection pool size, and a socket timeout. In the original class, there are at least four constructor methods to accommodate this situation. This condition is also called as the telescoping constructor, which is considered as an anti-pattern.

After applying the builder pattern, the number of the constructor methods of the IpgConfig class is reduced to one. The constructor class modifier is set to private, so that it cannot be constructed directly. To create the object, the builder static inner class is introduced. The mandatory parameters of the configurations are needed when creating the builder object. Then, optional parameters can be set one by one before creating the IpgConfig object. The builder pattern provides a better control over instantiation process.

Fig. 6 Builder design pattern implementation
4.3.2 Observer Design Pattern

The observer pattern, which is a behavioral design pattern, defines a one-to-many dependency between the subject and the objects so that when the subject changes state, the objects are notified and updated automatically. In this case study, the observer pattern could be implemented on handling the payment gateway provider notifications, which is conducted by the BusinessSpi as the subject. Firstly, a SPI observer abstraction is created. Thus, the classes that inherits it should realize its methods in their own way. Secondly, a list of the observers is added into the BusinessSpi, so the observers will be notified and updated when the BusinessSpi receives the notifications.

The example is shown by Fig. 7. There are two classes that inherits the observer, the OrderDao (data access object) and the CustomerNotifier class. These classes are added in the observer list of the BusinessSpi. When these classes are notified about the changes of an order status, the OrderDao will modify the transaction status in the database and the CustomerNotifier will send an email notification to the customer. By doing so, the subject and the observers are loosely coupled. The subject does not need to know about the implementation of the concrete observer class and any new observer can be added at any point of time without changing the subject class.

4.3.3 Factory Design Pattern

Like the builder pattern, the factory pattern is also a creational design pattern. It is used to create objects without exposing the instantiation process. In this case study, the factory pattern could be used when creating the API client object. As can be seen in the PSM, the RestApiClient is the only class that has responsible in posting a request to the payment gateway. However, this situation could be changed in the future. For instance, in the current situation the RestApiClient uses the technology of Apache
HttpClient. As an alternative way, a modern HTTP client, namely OkHttp by Square, Inc., is added. Thus, the OkHttpApiClient class is created. To improve the software design, an abstraction of API client, namely ApiClient, is introduced so that the RestApiClient and the OkHttpApiClient class inherits it. To create the concrete class of the abstraction, the ApiClientFactory class is needed. The factory design pattern makes the source codes more robust, less coupled, and easy to extend.

```java
ApiClient client = ApiClientFactory.createClient(config, 1);
BusinessApi api = new BusinessApi(client);
```

Fig. 8 Factory design pattern implementation
5 Analysis

In this chapter, five software metrics are measured by using static analysis to investigate the impact of the design patterns implementation. At first, the software metrics are measured when PSM is directly transformed into the source codes before (B) applying the design pattern. After (A) implementing the patterns, the metrics are evaluated again to analyze the differences. The SourceMeter [15] is used to measure these metrics. The metric categories that are chosen in this research, namely:

1. Cohesion metric, which is represented by the LCOM5 (Lack of Cohesion in Methods 5), in measuring to what extent the source code elements are coherent in the system.
2. Complexity metric, which is represented by the WMC (Weighted Methods per Class), in measuring the complexity of source code elements.
3. Coupling metric, which is represented by the CBO (Coupling Between Object classes), in measuring the amount of interdependencies of source code elements.
4. Inheritance metric, which is represented by DIT (Depth of Inheritance Tree), in measuring the aspects of the inheritance hierarchy of the system.
5. Size metric, which is represented by the LLOC (Logical Lines of Code), in measuring the basic properties of the analyzed system in terms of cardinalities.

<table>
<thead>
<tr>
<th>Table 1 Metrics measurement before and after applying design pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>B A</td>
</tr>
<tr>
<td>IpgConfig</td>
</tr>
<tr>
<td>IpgConfig$Builder</td>
</tr>
<tr>
<td>BusinessSpi</td>
</tr>
<tr>
<td>BusinessApi</td>
</tr>
<tr>
<td>RestApiClient</td>
</tr>
<tr>
<td>ApiClientFactory</td>
</tr>
</tbody>
</table>

The builder pattern decreases the complexity and the size of the IpgConfig class, but the coupling increases. It is because a static inner builder is introduced within the class. As can be seen in the Table 1, if the metrics of the IpgConfig class and the IpgConfig$Builder class are combined then the builder pattern makes the size and the complexity of the codes increasing. The observer pattern increases the cohesion of the BusinessSpi class since the LOM5 value decreases. However, the pattern also makes the complexity and the size of the class increasing. It is because the observers are introduced in the class. The factory pattern increases the coupling and the inheritance of the RestApiClient class because the abstraction and its factory class are introduced. Moreover, if the RestApiClient and the ApiClientFactory metrics are summed up, then the pattern also makes the complexity and the size of the source codes increasing.

6 Conclusion

The implementation of the selected design patterns in this case study increases the cohesion, the complexity, the coupling, the inheritance, and the size metrics of the source codes when they are introduced in the early stage of the software development. However, it could not be concluded if this is a good or a bad thing since the measured metrics might become tradeoffs to other software quality properties such as flexibility, maintainability, expandability, reusability, or understandability. More advantages or disadvantages of applying the design decisions might come later in the long run when
the software evolves over the time. For the future works, there should be a certain mechanism to judge whether applying a design pattern in such a case will give more positive or more negative impacts on each of the software quality attributes.

References