Consistent and Secure Transaction System Pattern

Ngo Huy Bien
Faculty of Information Technology, University of Science Ho Chi Minh City, Ho Chi Minh City, Vietnam
227 Nguyen Van Cu, District 5, Ho Chi Minh City, Vietnam
+84988437323
nhbien@fit.hcmus.edu.vn

Tran Dan Thu
Faculty of Information Technology, University of Science Ho Chi Minh City, Ho Chi Minh City, Vietnam
227 Nguyen Van Cu, District 5, Ho Chi Minh City, Vietnam
+84902765432
tdthu@fit.hcmus.edu.vn

ABSTRACT

Transaction processing is essential for all enterprise systems. In this paper, we propose a design pattern for modeling domain knowledge consistently and handling complicated transactions of applications that require enforcing security policies, especially the service-oriented and cloud-based applications in which each transaction is designed as a service and may invoke external services. We evaluated this design pattern by implementing it in our cloud-based enterprise systems, then executing business test cases and testing system performance. We also compared effort to understand a system designed using this pattern with another similar system designed using unstructured object-oriented design method and realized that this pattern reduces our developers' effort to understand the system. We hope that our consistent and secure transaction system pattern will be useful for software providers as well as business organizations when building service-oriented and cloud-based enterprise systems.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Architectures –*Patterns.*

General Terms

Design, Experimentation, Security.

Keywords

Transaction processing; Transaction pattern; Security pattern; Enterprise system; Domain-driven design; Service-oriented architecture; Cloud computing.

1. INTRODUCTION

A business transaction is an interaction in the real world, usually between an enterprise and a person or another enterprise, where something is exchanged. A business transaction is often a set of related tasks that lead to a particular goal. For example, in order to process an order for retail goods a system will need to perform the following tasks:

- Check the customer's credit, reserve the required material from stock, and schedule the shipment;
- Give commission credit to the salesperson; submit a request for payment from a credit card company;
- Perform the shipment, and then validate that the order was delivered.

Transaction processing is the processing of business transactions by computers connected by computer networks (Bernstein & Newcomer, 2009). Transactions processing is essential part of all enterprise systems. Implementing transactions for enterprise systems requires significant work due to complexity of domain and technologies. In this paper, we propose a model for consistently designing transaction processing of enterprise system in service-oriented and cloud-based environments. We also show that how security patterns (authentication and authorization) can be applied to transaction processing to enforce enterprise security policies.

The contributions of this paper lie in (i) a real-world requirements analysis of secure enterprise transaction processing; (ii) a design pattern to build transaction processing model for enterprise systems; (iii) a result from applying the design pattern in real word systems (evaluation of the design pattern). The pattern would enable both software providers as well as business organizations to reduce costs and time when building their service-oriented and cloud-based enterprise systems for automating their business transactions.

This paper is organized as follows: Section II discusses related work and challenges when building transaction applications. Section III presents our design pattern for transaction applications including its motivation, analysis, design, implementation, consequences and so on. Section IV presents evaluation of our design pattern. Finally, section V wraps up the paper and identifies a series of future tasks.

2. RELATED WORK

Transaction processing has been drawn much attention of researchers. In [3], the authors presented principles of transaction processing but they did not describe how to design and build a transaction processing application, especially in service-oriented or cloud-based environments. We presented a complete consistent design and guidelines to implement secure transactions for cloud-based applications.

Eric Evans presented a set of patterns (especially Repository and Factory pattern) for tacking complexity of domain [6]. However the author did not discuss how to enable configuration and security for enterprise transactions. The author also did not describe how to separate business rules from data access. We extended Repository and Factory patterns by combining with enforcing security polices (authentication, authorization and business security rules checking) and separating data access from business rules of transaction.

Mark Grand presented 4 related patterns (ACID Transaction, Composite Transaction, Audit Trail and Two Phase Commit) for handling transactions [10]. None of these patterns relate to security. These patterns also do not tackle complexity of domain knowledge. These patterns can play as tactics for implementing some aspects of our consistent and secure transaction system pattern (e.g. ACID properties).

Authentication model was proposed in [13]. Role-based access control models were proposed in [16]. Attribute-based access control model was introduced in [5]. These models play as key roles for implementing security for computer systems. These models play as tactics for implementing authentication and authorization aspects of our consistent and secure transaction system pattern.

Security patterns were also studied in depth in [20], [17], [12] and [7]. However there was little explanation about how to use them within a transaction system. In this paper, we show where security patterns can be applied within transaction processing to achieve authentication, authorization and security policies for a transaction.

Authentication aspect of a transaction system was addressed by applying an extra authentication step to those transactions considered sensitive because of their privacy requirements or monetary value in [1]. Our secure transaction system pattern is similar to this work in the sense of authentication. However we are different in that we handle additional tasks like configuration, authorization, business security policies and other business domain related tasks of a secure transaction system. Transaction authentication is only a part of our secure transaction system pattern. We are also different in that we suggest how to use external services or local persistence information for the system authentication. We are also different in that we propose a consistent design, in which all domain objects have the same representational form (i.e. Factory, Repository, Persistence entities).

3. CONSISTENT AND SECURE TRANSACTION SYSTEM PATTERN

This section describes our pattern for hierarchical multi-tenancy applications in GoF template [9].

3.1 Name

Consistent and Secure Transaction System Pattern.

3.2 Intent

Provide a consistent and flexible model for handling secure transactions of a software system.

3.3 Motivation

Each enterprise system must implement transactions to automate business processes. One key requirement of designing enterprise system transaction model is that transaction processing model should be flexible and tackle the complexity of domain knowledge. When we talk about domain knowledge, we mean domain objects. They are things like customer, user, role, authorization and so on. In other words, the transaction processing model should be able to handle complicated business transactions with consistent components and classes. It means that we should use similar steps and models when adding new business data or new business rules to our system.

One example is a company providing project management system software as a service[1]. This system contains many similar domain objects like *Customer*, *User*, *Role*, *Authorization*, *Project*, *Project Template*, *To-Do List*, *To-Do*, *Event*, *File*, *Message*, *Time Log*, *Risk*, and so on. The relationships among domain objects are complicated. There is a need of a consistent way to manage these domain objects with their business rules and data.

There is also a need of easily and consistently adding new domain objects that depend on specific customers to this system. For example construction project management may also require other domain objects like *Supplier*, *Contractor*, *Order*, *Material*, *Expense*, and so on.

Another requirement of enterprise system transactions is that each transaction should be executed in a secure manner. In other words, the transactions between enterprise users and system require enforcing security policies.

Transaction security can be achieved at different layers of a system (e.g. transport layer, application layer or storage) and by leveraging IT infrastructure, policies, etc. In this paper, we are only interested in enforcing security policies at application layer. We call *secure transaction* a transaction that requires authentication, authorization and enforcing other business security policies.

Let's take a look at Gmail system[2]. When a user wants to view a message, she sends a request to system. This request contains her credentials and message identifier. Once Gmail system receives request, it validates user's credentials. Then it validates account status against system policies. Then it checks if user has privileges to view the message. If everything is fine then it gets message content and returns it to user.

Let's return to the company providing project management system software as a service[1] for another example. The company provides a web page for users to buy their service. In order to buy a subscription, user sends a request to system. This request contains her credentials and other information for her order. Once system receives request, it validates user's credentials. Then it checks if user already has another subscription and other business rules. If everything is fine then it processes payment for user, saves user's subscription to system. Then it prepares product (i.e. services, storage) for user, notifies user about successful transaction, and displays transaction result to user.

How do we design transaction processing model for enterprise system so that domain knowledge can be managed consistently and all transactions are executed securely?

In order to solve this problem, we propose component-based enterprise architecture for secure transaction processing. Then we focus on design of specific components and explain how to achieve secure transaction processing with these specific components. This design pattern also provides consistency of components and classes for system to be extended. We also provide implementation notes for adapting pattern to various business cases.

Building on the component view of the enterprise architectures in [19], [11], [18] and our real world projects, our component-based

[1] https://basecamp.com

[2] https://mail.google.com

enterprise architecture for secure transaction processing is proposed in Fig. 1.

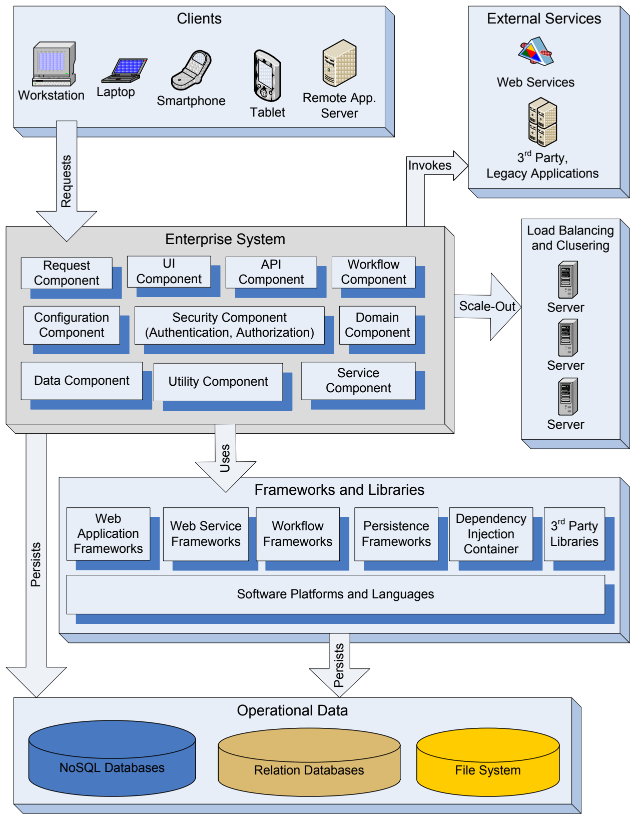

Figure 1. Enterprise System Architecture

A request from a *Client* is sent to *Request Component*. This is often a Uniform Resource Identifier (URI) to a Web User Interface (Web UI) or an Application Programming Interface (API). *Request Component* calls *Configuration Component* to get predefined settings for handling request. After that *Request Component* calls *Security Component* to authenticate and authorize the request.

After that *Request Component* calls *Domain Component* to process business request. After that *Request Component* calls *User Interface (UI) Component* or *API Component* to format result and returns result to *Client*. *UI Component* is also responsible for handling theme or branding for each request.

During execution, *Request Component*, *Configuration Component*, *Security Component* and *Domain Component* can call *Data Component* to get or save system state.

Utility Component can be called by any other components to perform specific processing that does not relate to domain knowledge. *Service Component* can be invoked when system needs to call 3rd party services (e.g. payment gateway, location service, notification service, and so on), 3rd party applications (e.g. email service, big data processing service, and so on) or legacy applications.

If the request needs reactive action, *Request Component* can pass the request to *Workflow Component* which is responsible for handling request and wait for stimulus from an external person or program. *Workflow Component* relies on other components to process the request.

All the components can use frameworks or libraries for their implementation or separation. The frameworks can be web application frameworks (e.g. ASP.NET MVC, Spring MVC, Ruby on Rails, and so on), web service frameworks, Workflow frameworks (e.g. Windows Workflow Foundation, Activiti, and so on), persistence framework (e.g. Hibernate, Entity Framework, and so on), dependency injection container (e.g. Spring, Castle Windsor, Unity Application Block, and so on), 3rd party libraries (e.g. UI libraries, AJAX libraries, logging libraries, caching libraries and so on). The frameworks and libraries in turn can be implemented using specific software platforms or languages (e.g. .NET, Java, PHP, Ruby, Python, Google App Engine, Force.com, Heroku, Eccentex's AppBase, and so on).

System states can be stored in various data stores. They may be relational databases (e.g Oracle, MS SQL, MySQL, Azure SQL Database), file system (e.g. text file, binary file) and NoSL database (e.g. Amazon S3, Google File System, Azure SQL, and so on).

The components can be implemented using Layers pattern [4], deployed using multi-tier architecture and scaled out for better performance.

Request Component, *UI Component*, *API Component*, and *Workflow Component* are often parts of software frameworks and libraries. When building an enterprise system we often extend software frameworks and libraries for these 4 components to match them with corporation business data and business processes.

Designing enterprise transactions involves configuration, security, domain knowledge, and data persistence and so on. In this paper, we are interested in designing objects inside *Configuration Component*, *Security Component*, *Domain Component*, *Data Component*, *Utility Component* and *Service Component* in a consistent way. The enterprise architecture and components are the context in which our pattern lies in. Our pattern captures a solution for recurring problem of consistently designing enterprise transaction systems.

We are also interested in providing authentication, authorization and business security policies for *secure transaction* using these proposed objects inside the components. We proposed a pattern called "Consistent and Secure Transaction System Pattern" for these two objectives. Our pattern is a composite pattern which composes of some known patterns [14]. This composite pattern solves a recurring problem about consistency, authentication and authorization in designing every enterprise transaction system.

3.4 Applicability

Use consistent and secure transaction system pattern when

- You want to reduce complexity of managing domain knowledge;
- You need to reuse domain knowledge for other system layers;
- You want to be able to extend a system in a consistent way;
- You want to separate domain knowledge from data access;
- You want to enforce security policies (authentication, authorization and other business security policies) when handling transactions.

3.5 Structure

A typical secure transaction design pattern UML object structure is described in Fig. 2. For simplicity, we remove some similar interfaces and objects from diagram (e.g. *IEntity2Factory*, *Entity2Factory*, *IEntity3Factory*, *Entity3Factory*, *IEntity4Factory*, *Entity4Factory*, *IEntity4Repository*, *Entity4Respository*, *IEntity4Persistence* and *Entity4Persistence*).

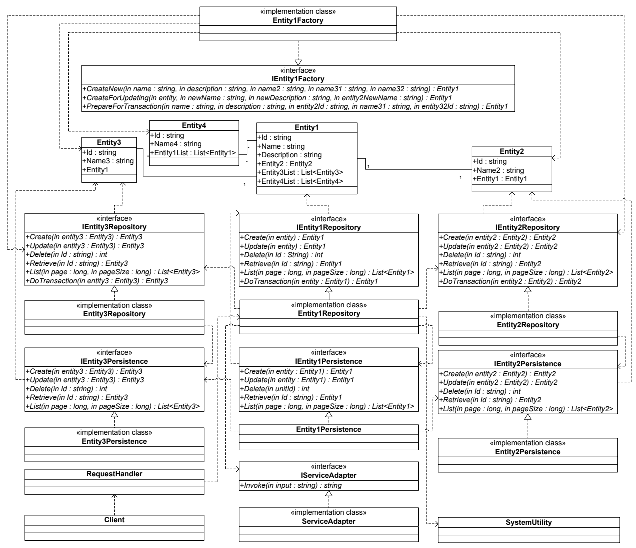
«implementation class»
Entity1Factory
«interface»
IEntity1Factory
+CreateNew(in name : string, in description : string, in name2 : string, in name31 : string, in name32 : string) : Entity1
+CreateForUpdating(in entity, in newName : string, in newDescription : string, in entity2NewName : string) : Entity1
+PrepareForTransaction(in name : string, in description : string, in entity2Id : string, in name31 : string, in entity32Id : string) : Entity1
Entity4
+Id : string
+Name4 : string
+Entity1List : List<Entity1>
Entity3
+Id : string
+Name3 : string
+Entity1
Entity1
+Id : string
+Name : string
+Description : string
+Entity2 : Entity2
+Entity3List : List<Entity3>
+Entity4List : List<Entity4>
Entity2
+Id : string
+Name2 : string
+Entity1 : Entity1
1
1
1
«interface»
IEntity3Repository
+Create(in entity3 : Entity3) : Entity3
+Update(in entity3 : Entity3) : Entity3
+Delete(in Id : string) : int
+Retrieve(in Id : string) : Entity3
+List(in page : long, in pageSize : long) : List<Entity3>
+DoTransaction(in entity3 : Entity3) : Entity3
«interface»
IEntity1Repository
+Create(in entity) : Entity1
+Update(in entity) : Entity1
+Delete(in Id : String) : int
+Retrieve(in Id : string) : Entity1
+List(in page : long, in pageSize : long) : List<Entity1>
+DoTransaction(in entity : Entity1) : Entity1
«interface»
IEntity2Repository
+Create(in entity2 : Entity2) : Entity2
+Update(in entity2 : Entity2) : Entity2
+Delete(in Id : string) : int
+Retrieve(in Id : string) : Entity2
+List(in page : long, in pageSize : long) : List<Entity2>
+DoTransaction(in entity2 : Entity2) : Entity2
«implementation class»
Entity3Repository
«implementation class»
Entity1Repository
«implementation class»
Entity2Repository
«interface»
IEntity3Persistence
+Create(in entity3 : Entity3) : Entity3
+Update(in entity3 : Entity3) : Entity3
+Delete(in Id : string) : int
+Retrieve(in Id : string) : Entity3
+List(in page : long, in pageSize : long) : List<Entity3>
«interface»
IEntity1Persistence
+Create(in entity : Entity1) : Entity1
+Update(in entity : Entity1) : Entity1
+Delete(in unitId) : int
+Retrieve(in Id : string) : Entity1
+List(in page : long, in pageSize : long) : List<Entity1>
«interface»
IEntity2Persistence
+Create(in entity2 : Entity2) : Entity2
+Update(in entity2 : Entity2) : Entity2
+Delete(in Id : string) : int
+Retrieve(in Id : string) : Entity2
+List(in page : long, in pageSize : long) : List<Entity2>
«implementation class»
Entity3Persistence
Entity1Persistence
«implementation class»
Entity2Persistence
RequestHandler
«interface»
IServiceAdapter
+Invoke(in input : string) : string
«implementation class»
ServiceAdapter
Client
SystemUtility

Figure 2. Consistent and Secure Transaction System Pattern Static Structure

3.6 Participants

Client represents client requests processing for a transaction. This is often an object from *Request Component* or *UI Component* or *API Component* or *Workflow Component*.

RequestHandler represents an object that is responsible for receiving inputs for transaction from *Client* and returns outputs to *Client*.

Entity1 represents any entity in system. This could be any domain objects, including *User*, *Role*, *Company*, *Authorization*, *Configuration*, and any *Business Entities (e.g. Product, Order, OrderItem, Payment, Address, Invoice and so on)*.

Entity2 represents any entity in system that has one-to-one relationship with *Entity1*.

Entity3 represents any entity in system that has many-to-one relationship with *Entity1*.

Entity4 represents any entity in system that has many-to-many relationship with *Entity1*.

IEntity1Repository defines interfaces for working with *Entity1* object. This interface represents all objects of *Entity1* type as a conceptual set. This interface is responsible for handling all domain knowledge of *Entity1* object.

Entity1Repository represents an implementation of *IEntity1Repository* interface.

IEntity1Persistence defines interfaces for working with persistence of *Entity1*. This interface is responsible for handling all data operations against *Entity1*.

Entity1Persistence represents an implementation of *IEntity1Persistence* interface.

IEntity1Factory defines interfaces for encapsulation for *Entity1* object creation, especially when creation of *Entity1* object becomes complicated or reveals too much of the internal structure.

Entity1Factory represents an implementation of *IEntity1Fatory* interface.

IServiceAdapter defines interfaces for encapsulation for working with an external service. If system needs to interact with many external services then there will be *IServiceAdapter2*, *IServiceAdapter3* and so on.

ServiceAdapter represents an implementation of *IServiceAdapter* interface. If there are *IServiceAdapter2*, *IServiceAdapter3* and so on then there will be *ServiceAdapter2*, *ServiceAdapter3* and so on.

SystemUtility represents object providing helper functions for system. These functions do not belong to any specific domain knowledge. These functions may be string processing functions, date and time processing functions, cryptography functions and so on. If system has many different utilities then there will be *SystemUtility2*, *SystemUtility3* and so on.

IEntity2Repository defines interfaces for working with *Entity2*. This interface represents all objects of *Entity2* type as a conceptual set. This interface is responsible for handling all domain knowledge of *Entity2* object. This interface can be used by *EntityRepository* when *EntityRepository* needs to access domain knowledge of *Entity2* object.

Entity2Repository represents an implementation of *IEntity2Repository* interface.

IEntity2Persistence defines interfaces for working with persistence of *Entity2*. This interface is responsible for handling all data operations against *Entity2*.

Entity2Persistence represents an implementation of *IEntity2Persistence* interface.

IEntity2Factory defines interfaces for encapsulation for *Entity2* object creation, especially when creation of *Entity2* object becomes complicated or reveals too much of the internal structure. This interface can be used by *EntityFactory* when *EntityFactory* needs to create an instance of *Entity2* object for *Entity.*

Entity2Factory represents an implementation of *IEntity2Fatory* interface.

IEntity3Repository and *IEntity4Repository*, *Entity3Repository* and *Entity4Repository*, *IEntity3Persistence* and *IEntity4Persistene*, *Entity3Persistence and Entity4Persistence*, *IEntity3Factory* and *IEntity4Factory*, *Entity3Factory* and *Entity4Factory* play similar roles as *IEntity2Repository, Entity2Repository, IEntity2Persistence, Entity2Persistence, IEntity2Factory* and *Entity2Factory*, respectively.

3.7 Collaborations

In figure 2, we propose a template for designing all domain objects. We will describe collaborations of the system objects in a more concrete view. Figure 3 describes process of executing a secure transaction for placing an *Order*, i.e. a concrete domain object. A note is that figure 2 and figure 3 do not have the same level of abstraction.

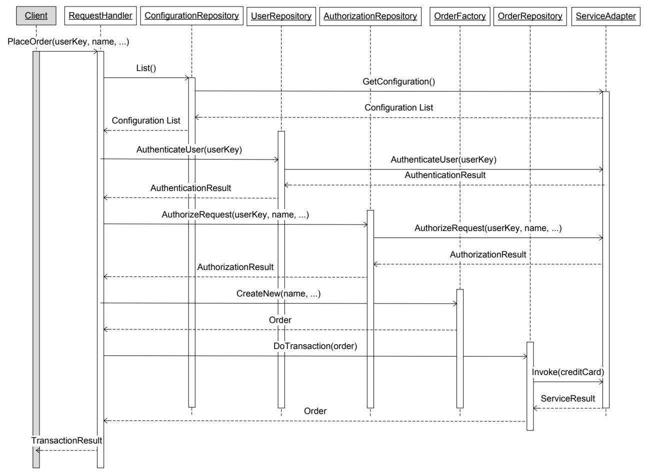

Figure 3. Consistent and Secure Transaction System Pattern Sequence Diagram

When *Client* sends a request to system, the request is captured by *RequestHandler*. This is entry point of transaction. The transaction begins to be executed. *RequestHandler* calls *ConfigurationRepository* to retrieve system configurations**.** We recall that *Configuration* is one of system domain objects (*Enitity1* or *Entity2* or *Entity3* or *Entity4* or other entity). *ConfigurationRepository* is responsible for handling system configurations, e.g. global application settings, security policies. These configurations can be applied to all transactions, including current transaction. *ConfigurationRepository* can also returns specific configurations based upon request's specific parameters or optionally call *ServiceAdapter* to get configurations from external services.

After that *RequestHandler* calls *UserRepository* to authenticate request**.** *UserRepository* is responsible for handling system authentication, e.g. credentials validation, security policies validation, account status checking. We recall that *User* is one of system domain objects. *UserRepository* is responsible for managing domain knowledge of *User***.** *UserRepository* can call *UserPersistence* to authenticate *User* using the system data store. *UserRepository* can optionally call *ServiceAdapter* to invoke external services for authentication, e.g. single sign on service.

After that *RequestHandler* calls *AuthorizationRepository* to authorize request. *AuthorizationRepository* is responsible for handling system authorization, e.g. resource security policies validation, account privileges checking. We recall that *Authorization* is one of system domain objects and *AuthorizationRepository* is responsible for managing domain knowledge of *Authorization. Authorization* object is often an aggregate of *User* or *Role* and *Resource* and *Action***.** *AuthorizationRepository* can call *AuthorizationPersistence* to authorize request using the system data store. *AuthorizationRepository* can optionally call *ServiceAdapter* to invoke external services for authorization.

After that *RequestHandler* calls *OrderFactory* to create an *Order* for business processing. *RequestHandler* calls *OrderRepository* to place an order. During this execution, *OrderRepository* may call *ConfigurationRepository*, *UserRepository*, *AuthorizationRepository*, *SystemUtility*, other domain *Repositories* (e.g. *OrderItemRepository*, *AddressRepository*, and so on) or *Factories* (e.g. *OrderItemFactory*, *AddresssFactory*, and so on) to get information for executing transaction. *OrderRepository* may call *ServiceAdapter* to invoke external services for processing transaction (e.g. processing payment, sending notification saving system state). *OrderRepository* may call *OrderPersistence* to get or save system state.

3.8 Consequences

The advantage of this pattern is that it reduces complexity when managing domain knowledge of an enterprise system. It provides a consistent approach for modeling domain knowledge and handling transactions. The consistency is achieved by representing all domain objects in the same form. Figure 2 shows that each domain object is represented by *Repository*, *Factory*, and *Persistence* object. Figure 2 acts like a template for designing the domain objects.

Because all objects and interfaces are represented in a consistent form (please refer to figure 2) and all secure transactions are handled in the same way (please refer to figure 3) the system complexity will be reduced. In other words it makes developers easier to understand and extend the system or create a new system following this solution. One important note is that the pattern may contain more objects. However adding more objects and interfaces in this consistent way will not make system more complex.

Another advantage of this pattern is that it separates business data access from business rules. This is done by encapsulating business data access into *Persistence* object and encapsulating business rules into *Repository* and *Factory* object. This will make system easier for maintenance.

This pattern also shows where security aspects can be applied in transaction processing. The security aspects on which the pattern focuses are authentication, authorization and security policies. Authentication tasks are encapsulated in *UserRepository* and *ServiceAdapter* object. Authorization tasks are encapsulated in *AuthorizationRepository* and *ServiceAdapter* object. Security policies can be encapsulated in any system object, for example *OrderRepository*.

3.9 Implementation

There are some important points that need to be considered when implementing consistent and secure transaction system pattern:

- Any entity (*Entity1*, *Entity2*, *Entity3*, *Entity4* and so on) can be implemented using Composite pattern [9] when hierarchy is required. The number of entities depends on business data and business processes. These entities are often grouped into a layer and form a business model of system.
- System entry point (*RequestHandler*'s responsibility) can be implemented using Single Access Point pattern [20].
- Request handling (*RequestHandler*'s responsibility) can be implemented using Command pattern [9] when requests need to be logged for later processing.
- Request handling can also be implemented using Intercepting Filter pattern [2] when a request and a response are needed to be manipulated before and after the request is processed.
- Entry points of domain services (*RequestHandler*'s responsibility) can be implemented using Façade pattern [9] when coarse-grain services are needed.
- Wrappers for external services (*ServiceAdapter*'s responsibility) can be implemented using Adapter pattern

[9] when interfaces of enterprise system and external services are incompatible.

- Wrappers for external services (*ServiceAdapter*'s responsibility) can also be implemented using Strategy pattern [9] when system needs the ability to switch among external services. All service adapter interfaces and classes are usually put into separate layers for easier later changes.
- Entity persistence (*Entity1Persistence*'s, *Entity2Persistence*'s, *Entity3Persistence*'s, *Entity4Persistence*'s responsibility) can be implemented using Strategy pattern [9] when enterprise system needs the ability to switch among different data stores. All persistence interfaces and classes are usually put into separate layers for easier later changes.
- Entity authorization (*AuthorizationRepository*'s responsibility) can be implemented using Check Point pattern [20] and/or RBAC pattern [7] and/or ABAC pattern [5].
- ACID properties can be achieved using language or platform specific features (e.g. System.Transactions namesapce of .NET Framework) and Unit of Work pattern [8] or ACID Transaction pattern [10].

3.10 Sample Code

The following fragment C# code implements a part of transaction pattern using C# generic feature.

```
// A generic interface is defined
// for all entity repositories
public interface
IEntityRepository<T>{
   T Create(T entity);
   T Retrieve(Guid entityId);
   int Update(T entity);
   int Delete(Guid entityId);
}

// This interface defines specific
// methods for OrderRepository
public interface IOrderRepository{
   Guid CheckOutOrder(Order order,
   String notificationTemplate,
   AppSettings appSettings);
}

public class OrderRepository :
IOrderRepository,
IEntityRepository<Order>{
 private IEntityPersistence<Order>
             orderPersistence;
 public OrderRepository(){
   this.orderPersistence =
       new OrderPersistence ();
 }

 public Order Create(Order order){
   Checker.Require(null != order,
       "Order cannot be null.");
    Checker.Require(
       null != order.Customer,
       "Customer cannot be null.");
   Checker.Require(
      null != order.PaymentMethod,
      "Payment method cannot be null.");
   return
   this.orderPersistence.Create(order);
 }

 public Guid CheckOutOrder(Order order,
 String notificationTemplate,
 AppSettings appSettings){
    String resultCode = String.Empty;
    String resultMessage = String.Empty;
    PaymentGatewayHelper.Checkout(
    order, out resultCode,
    out resultMessage);
    return Order.OrderId;
 }
}
```

3.11 Known Uses

- This pattern was used to construct transaction processing for our secure messaging system, project management system, corporate information system and school management system.
- Most current enterprise system transaction implementation shares the idea of this pattern.

3.12 Related Patterns

Repository pattern [6], Factory pattern [6], ACID Transaction [10], Data Access Object [2].

4. EVALUATION

We evaluated instances of our (candidate) pattern using case studies in order to realize if our pattern can produce transaction systems satisfying our customers' business needs of secure transactions and if our pattern can reduce developers' effort to understand the systems [15]. We also evaluated response time of transaction processing when all system transactions are designed using this pattern.

The pattern was used to construct a core framework that managing tenants, users, roles, authorization, web pages, modules, messages, folders, files, and so on. All these entities are managed in a consistent manner. Then our complicated business processes, e.g. handling customers' subscriptions, handling customers' orders, managing corporation events, meetings, sending messages securely, sharing files and so on were handled by extending the framework and following the pattern collaboration instructions. All business transactions were tested by our customers successfully.

We compared effort to learn a system designed using this pattern with an old similar system designed using an inconsistent way. It took our new developers less effort to understand the system designed using this pattern.

We used ASP.NET 4.0 for our system front-end and MSSQL 2008 server for our system back-end when implementing this pattern. Our system was deployed on 5 Intel Xeon servers (2 web servers with load balancing, 1 application server, 1 database server and 1 mail server). Each server has 3GHz CPU (2 processors) and 6GB RAM. In our current SaaS secure messaging system, the number of tenants of is about 5000, the number of users is about 125000, the number of messages is about 2626400 and the number of files is about 1577000. System response time of a transaction of getting an authorized message with metadata of 3 attached files is less than 3 seconds when there are simultaneous 100 requests from Load Impact[3].

5. CONCLUSION AND OUTLOOK

Designing transaction processing model for enterprise systems takes a lot of time and efforts. In this paper, we have discussed the requirements of transaction processing in real world applications and proposed a solution for building transaction processing model for enterprise systems, especially in service-oriented or cloud-based environments. We have presented the results in a pattern form so that it can be adapted to different systems. We hope that our consistent and secure transaction system pattern will be useful for building enterprise systems for software providers as well as business organizations.

Future work includes extending this pattern to handle secure business transaction that spans multiple requests (composite transaction), describing how to handle business transaction with web service standards, detailed explanation about concurrency control and recovery.

6. ACKNOWLEDGMENTS

We thank our shepherd, Eduardo B. Fernandez, for his valuable comments and suggestions that improved our paper.

7. REFERENCES

[1] A. BRAZ, FABRICIO; B. FERNANDEZ, EDUARDO; C. RISPOLI, DIOGO. Transaction Authentication Pattern. In *MiniPLoP Brazil 2013* (Brasília 2013).

[2] Alur, Deepak, Crupi, John, and Malks, Dan. *Core J2EE™ Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2001.

[3] Bernstein, Philip A. and Newcomer, Eric. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[4] Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, and Stal, Michael. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[5] C. Hu, Vincent, Ferraiolo, David, Kuhn, Rick, Schnitzer, Adam, Sandlin, Kenneth, Miller, Robert, and Scarfone, Karen. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations (DRAFT)*. 800-162, NIST, 2013.

[6] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, Boston, 2003.

[7] Fernandez, Eduardo B. *Security Patterns in Practice: Designing Secure Architectures using Software Patterns*. Wiley, Chichester, 2013.

[8] Fowler, Martin, Rice, David, Foemmel, Matthew, Hieatt, Edward, Mee, Robert, and Stafford, Randy. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[9] Gamma, Eric, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] Grand, Mark. Mark Grand. Transaction Patterns: A Collection of Four Transaction Related Patterns. In *PLoP '99* (1999).

[11] Kanalakis, John. *Developing.NET Enterprise Applications*. Apress, 2003.

[12] Kanneganti, Ramarao and Chodavarapu, Prasad. *SOA Security*. Manning Publications Co., Greenwich, 2008.

[13] NEUMAN, B. CLIFFORD; TS'O, THEODORE. Kerberos: An Authentication Service For Computer Networks. *IEEE Communications Magazine* (1994), 33-38.

[14] Riehle, Dirk. Composite Design Patterns. In *OOPSLA '97* (Atlanta 1997), ACM Press, 218–228.

[15] RUNESON, PER; HÖST, MARTIN. Guidelines For Conducting And Reporting Case Study Research In Software Engineering. *Empirical Software Engineering*, 14, 2 (Dec. 2009), 131-164.

[16] SANDHU, RAVI S.; J. COYNEK, EDWARD; FEINSTEINK, HAL L.; E. YOUMANK, CHARLES. Role-Based Access Control Models. *IEEE Computer* (1996), 38-47.

[17] SCHUMACHER, MARKUS; FERNANDEZ-BUGLIONI, EDUARDO; HYBERTSON, DUANE; BUSCHMANN, FRANK; SOMMERLAD, PETER. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons Ltd, Chichester, 2006.

[18] Shroff, Gautam. *Enterprise Cloud Computing: Technology Architecture Applications*. Cambridge University Press, 2010.

[19] Sowa, J.F. and Zachman, J.A. Extending and Formalizing the Framework for Information Systems Architecture. *IBM Systems Journal*, 31, 3 (1992), 590-616.

[20] Yoder, Joseph and Barcalow, Jeffrey. Architectural Patterns for Enabling Application Security. In *PLoP '97* (Monticello 1997).

[3] http://loadimpact.com/