Two new software behavioral design patterns: Obligation Link and History Reminder

Stefan Andrei
Lamar University, Beaumont, TX, U.S.A.
Stefan.Andrei@lamar.edu

Arnob Saha
Lamar University, Beaumont, TX, U.S.A.
ASaha@lamar.edu

Sojibur Rahman
Lamar University, Beaumont, TX, U.S.A.
SRahman2@lamar.edu

Abstract

Finding proper design patterns has always been an important research topic in the software engineering community. One of the main responsibilities of the software developers is to determine which design pattern fits best to solve a particular problem. Design patterns support the effort of exploring the use of artificial intelligence in better management of software development and maintenance process by providing faster, less costly, smarter and on-time decisions (Pena-Mora & Vadhavkar, 1996). There has been a permanent interest in finding new design patterns, especially in the last two decades. Many new design patterns apply in various areas of computer science, such as software security, software parallelism, **large-scale software evolving, artificial intelligence, and more**. To the best of our knowledge, the "Obligation Link" and "History Reminder" design patterns are new and can be applied in software development in many areas of computer science including artificial intelligence.

Keywords: behavioral design patterns, software engineering, artificial intelligence

1. Introduction

It is well known that a software design pattern is a general reusable solution of a commonly occurring problem within a given context in software design. In other words, it is a description or template for solving a software problem that can be used in many different situations. Software design patterns have always played a significant role in the software engineering community. One of the first de-facto reference for describing design patterns is (Erich, Helm, Ralph & Vlissides, 1997), published by Erich, Helm, Johnson, and Vlissides (also known as 'The Gang of Four'). This textbook published a list of 23 design patterns and many software development companies adopted these design patterns for developing their software. There are many other works including descriptions and applications of software design patterns, such as (Knox, 2011) (OODesign, 2015) (Pmsware, 2013) and more. Many textbooks (Priestley, 2004) (Larman, 2005) (Wadhwa, Andrei & Yuen Jien, 2007) and articles (Bashir, 2010) (Noborikawa, 2003) offer implementations in various object-oriented programming languages of the design patterns from (Erich, Helm, Ralph & Vlissides, 1997). In addition, there exist software tools that assist programmers in using the popular design pattern for a solution, such as Rational Rose (IBM, 1999), Argo UML (Tigris, 2005) and more. There are three main categories of design patterns (Erich, Helm, Ralph & Vlissides, 1997) (Noborikawa, 2003) (Shvets, A., Frey, G., Pavlova, 2010):

1. *Creational* Design Pattern: These patterns are used to create objects for a suitable class that serves as a solution for a problem. They are particularly useful when dealing with polymorphism.

2. *Structural* Design Pattern: These patterns form larger structures from individual parts. Structural patterns vary a great deal depending on the structure and they are concerned with classes and objects.
3. *Behavioral* Design Patterns: These patterns describe interactions between objects. They focus on how objects communicate with each other.

Researchers from the software engineering community continue to improvc the existing list of design patterns to make them more secure and safe. Dougherty et al. described six secure design patterns in a very similar way as in (Erich, Helm, Ralph & Vlissides, 1997), such as: Secure Factory, Secure Strategy Factory, Secure Builder Factory, Secure Chain of Responsibility, Secure Logger, and Clear Sensitive Information (Dougherty, Sayre, Seacord, Svoboda, & Togashi, 2009). Similarly, Keutzer and Mattson also described eight design patterns for parallel programming, such as: Dense Linear-Algebra, Pipe-and-Filter, Data Parallelism, Loop Parallel, Stream Processing, Map-Reduce, Geometric Decomposition, and Kernel Parallelism [Keutzer & Mattson, 2010). *Wei Wang et al. presented an application of design patterns in the process of large-scale software evolving (Wang, 2010).* Pena-Mora and Vadhavkar described design patterns for supporting the effort of exploring the use of artificial intelligence in better management of software development and maintenance process by providing faster, less costly, smarter and on-time decisions (Pena-Mora & Vadhavkar, 1996). Dill et al. proposed several design patterns that can be applied to the configuration of utility-based artificial intelligence (Dill, 2012).

Even if there were many contributions to writing and applying design patterns, we believe that there is still room for improvement of the existing design patterns. For instance, considering the well-known design pattern “Chain of Responsibility” (Erich, Helm, Ralph & Vlissides, 1997) an unexpected request may delay the execution of the entire program. In this kind of situation none of the chains can actually determine and solve the request. According to the semantics of the “Chain of Responsibility” design pattern, the request will be forwarded to every object. Having a large number of such unexpected requests would imply that the “Chain of Responsibility” is not a proper design pattern to be used. This paper describe an alternative design pattern able to handle a large number of requests. We called our proposed design pattern the “Obligation Link” design pattern that will be described in the rest of the article.

In order to describe our second new design pattern, let us consider the existing “Memento” design pattern (Erich, Helm, Ralph & Vlissides, 1997) (Sanders, 2014). The “Memento” design pattern uses three classes to manage one Memento object. For complex software design it is difficult to manage the complex Originator class. Having an Originator object, copying large amounts of data to store in the Memento object might lead to a considerable overhead. For the Caretaker class it takes a certain amount of time and space to manage a complex Caretaker object to retrieve the state of a Memento object. Our second contribution of this paper is to provide a solution that solves the existing difficulties of the Memento design pattern. We called our new second design pattern “History Reminder”.

To the best of our knowledge, the “Obligation Link” and “History Reminder” design patterns are new. The main contribution of our paper is to describe their applicability, benefits, and implementation in a Java-like programming language.

2. Background

Large scale software development may have a slightly different classification viewed in general as a refinement of the traditional design patterns classification from (Erich, Helm, Ralph & Vlissides, 1997). This is because a large project usually has many levels to be considered, such as algorithmic, computational, implementation, execution, and structural. The previous section illustrated the limitations of two traditional existing software design patterns and their implementation: “Chain of Responsibility” and “Memento”. As an alternative of these two patterns,

we proposed two new design patterns for large scale software development. Here is the classification in five levels of the software design patterns:

1. Algorithmic strategy patterns focus on high-level strategies and how to exploit application characteristics on a computing platform.
2. Computational design patterns are related to the key computation identification.
3. Execution design patterns address issues about lower-level support of application execution. It includes strategies for executing streams of tasks and building blocks to support task synchronization.
4. Implementation strategy patterns concern about implementing source code to support: program organization and common data structures specific to parallel programming.
5. Structural design patterns focus on global structure of application being developed.

The rest of the paper is organized as follows. Section 3 studies and analyzes the "Chain of Responsibility" design pattern together with our alternative, "Obligation Link" design pattern. Section 4 presents the "Memento" design pattern, as well as our alternative, "History Reminder" design pattern. Section 5 refers to experimental results about using these two new design patters. Conclusions and References end this paper.

3. The "Obligation Link" design pattern

This section describes the background, applicability, structure, participants, collaborations, consequences, and implementation of the "Obligation Link" design pattern.

3.1. Background

In writing a software application which uses the "Chain of responsibility" design pattern, it often happens that we have to take a decision in advance on whether the request made by the client is going to be handled or not. This decision can reduce the execution time of the request.

Example 1. Let us assume that we have designed an application that can perform addition, subtraction and multiplication operations. So, in the case of a "Chain of responsibility" design pattern if a client makes a request of say, a división, the client has to wait until the request goes to the end of the chain and find out that the application cannot actually perform a division operation. Hence, the application has to run lots of unnecessary code which will increase the execution time. ■

Our concern was if we can find a technique of knowing in advance that the application can perform division or not. In this way, we can save some execution time and use that time to handle some other requests from the same or different clients.

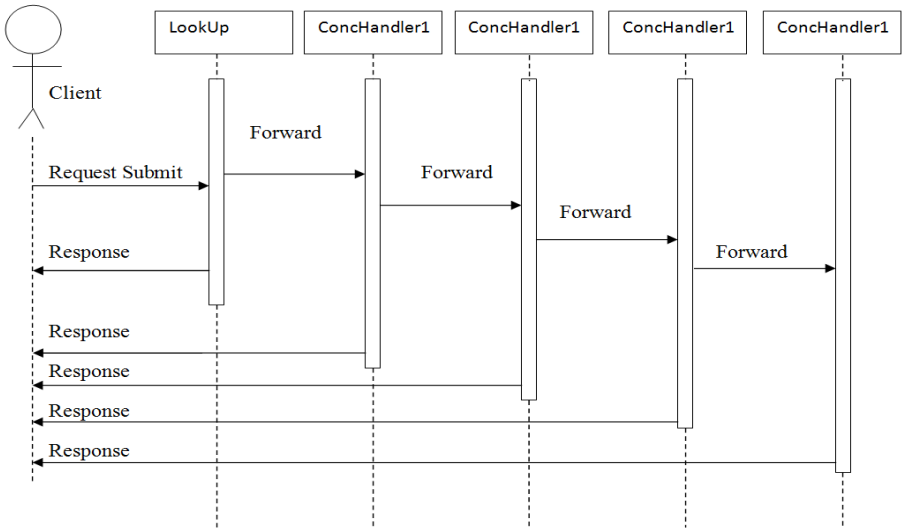

Figure 1. The sequence diagram of the "Obligation Link" design pattern

A *sequence diagram* is used to model the flow of the logic of the system. A sequence diagram represents perhaps the most popular Unified Modelling Language (UML) diagram that describes the behavior of a scheme. Figure 1 shows the sequence diagram of "Obligation Link" design pattern.

3.2. Applicability

The "Obligation Link" design pattern should be used in one of the following three software situations:

1. We need a decision in advance that the request is going to be handled or not;
2. We want to reuse the action attached to the handler;
3. We want to reduce some execution time because of the unexpected request.

3.3. Structure

Here is the structure of the "Obligation Link" design pattern described through another UML concept, called *class diagrams*. Contrary to the sequence diagrams which describe the behaviour, class diagrams specify static structural information about a program. The important information for a class, such as its *attribute*, *operation*, etc, along with relationship between classes can be specified using a class diagram.

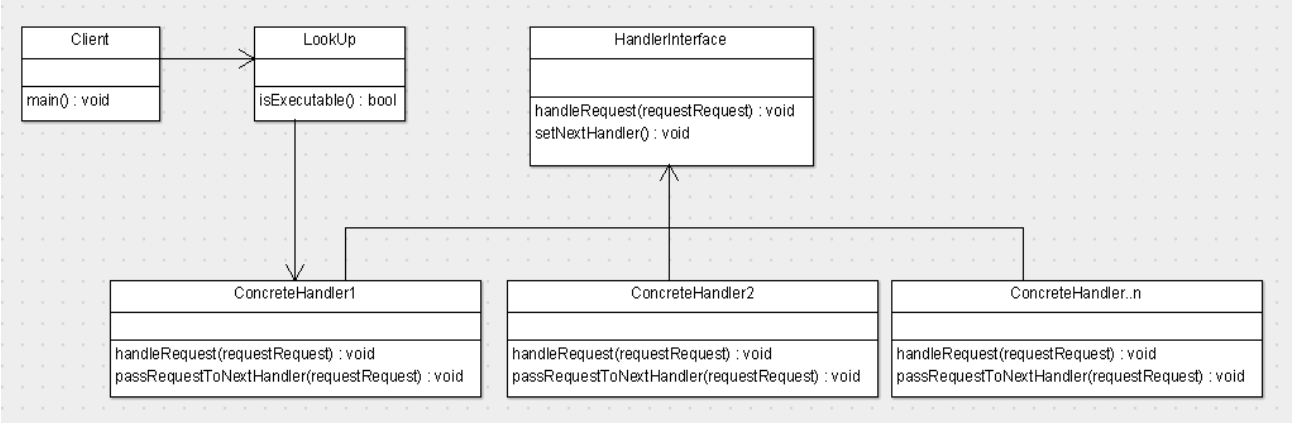

Figure 2. The class diagram of the "Obligation Link" design pattern

3.4. Participants

There exist four participants in the "Obligation Link" design pattern, such as: Client, LookUp, Handler, and ConcreteHandler. Here is the list of their responsibilities:

- ✓ Client
  - Initiates the request to the lookup class.
- ✓ LookUp
  - maintain a collection of objects
  - Search for the object responsible for the corresponding request.
  - Sends the request to the first concrete handler of the chain.
- ✓ Handler
  - defines an interface for handling the request
- ✓ ConcreteHandler
  - Handles the request it is responsible for
  - or sends the request to the next handler in the chain

3.5. Collaborations

When a client makes a request, it goes to the LookUp class. After searching the collection of a responsible object, the object of LookUp class sends the request to the first ConcreteHandler of the chain.

3.6. Consequences and benefits

The "Obligation Link" design pattern has the following benefits:

1. If the chain is properly configured then the request will be handled when it is in the chain;
2. This pattern can reduce the execution time by simply searching the collection when there is no object in the chain which is responsible for the specific request.

3.7. Implementation

This section is dedicated to describing a Java-like implementation of the "Obligation Link" design pattern.

3.7.1. Passing the request to the next handler

Handler keeps record to next handler to which it will pass the request. We can set the next handler property like below:

```
ConcreteHandler1 concrete1 = new ConcreteHandler1();
ConcreteHandler2 concrete2 = new ConcreteHandler2();
ConcreteHandler3 concrete3 = new ConcreteHandler3();
concrete1.NextHandler = concrete2;
concrete2.NextHandler = concrete3;
```

In this way, we can make a chain of the handler that will receive the request one by one until one of them executes the request.

3.7.2. Passing the request to the first handler

We have used an execute() method to pass the request to the first ConcreteHandler of the chain. The format looks like:

```
Request request = new Request();
firstConcreteHandler.Execute(request);
```

3.7.3. Executing an action related to the request

When we want to create some ConcreteHandler objects we just implement the Handler interface. The interface has some functionality that the client needs to implement in the successor class. The handler must implement the method execute method which checks whether the handler is going to complete the request or it will pass the request to the next handler. The default behavior looks like:

```
firstConcreteHandler.Execute(request);
public void executes(Request request) {
   if (someCondition) {
     ExecuteCorrespondingAction(request);
   }
    else {
     PassRequestToNextHandler(request);
    }
}
```

3.7.4. Adding Object to the Lookup collection

The Lookup object has a collection of Handler objects. We can add a Handler object to the collection object using an addObjectToCollection() method, such as the following code:

```
LookupClass someClass = new LookupClass();
someClass.addObjectToCollection(new ConcreteHandler());
```

3.7.5. The Lookup Class

When a client makes a request, it passes through the LookUp class. This class has an internal collection of the object where it stores all the object of the entire program.

```
public class LookUpClass {
  public boolean searchArray (String elementToSearch) {
    for (int i = 0; i < lengthofCollection; i++) {
      if (collection[i].equals(elementToSearch)) {
            return true;
      }
    }
    return false;
  }
}
```

4. The "History Reminder" Design Pattern

This section describes the intent, motivation, concept, participants, applicability, structure, collaborations, consequences, and implementation of the "History Reminder" design pattern.

4.1. Intent of the "History Reminder" design pattern

Designing a software application without implementation of a proper design pattern may lead to an incomplete one. Design patterns play an important role when considering the scalability of the software. By implementing the proper design pattern we can reuse the existing code with minimal complexity of time and space. Applying one of the behavioral design patterns (Kanasz, 2013) named "Memento" for large software applications, some implementation difficulties may arise from certain implementation. In addition, large software applications may lead to a considerable amount of cost and time for implementing the "Memento" design pattern. We propose an alternative solution by improving the difficulties and shortcomings of the "Memento" design pattern. We propose our new design pattern named as "History Reminder" design pattern.

4.2. Motivation of the "History Reminder" design pattern

There are many ways to organize the design patterns depending on their structure and behavior. In this research work, we investigate the previous design patterns purposes and scopes for the software applications. Based on the design patterns granularity and the level of abstraction, we decided to classify the design patterns in groups. Among all other dynamic design patterns, we have focused on one of the popular design patterns "Memento" design pattern. We reveal the concept, limitations and applications of this design pattern.

4.2.1. The concept of "History Reminder" design pattern

The "History Reminder" design pattern is based on the history of an object. It hides the HistoryReminder class from other classes and saves the data as state of HistoryReminder object. It does not violate the data encapsulation rule of object oriented software design.

The key concepts of the "History Reminder" design pattern are given below:

1. It keeps client class data encapsulated and create checkpoint of the each state.

2. Only the Originator class can access and create the History class. Checkpoints have been created within the History class. Also with the help of the Originator, the History class restores itself without taking any help of third party classes (the Caretaker class).
3. Information has been saved as the form of state of the HistoryReminder class. Every time the information is stored, it defines a state and increase the current state by one.

4.3. Participants and their semantics of the "History Reminder" design pattern

The History design pattern consists of two main clases: the HistoryReminder class and the HistoryReminderOriginator class. We describe below the semantics of these two classes as well as their association.

4.3.1. The HistoryReminder Class

The HistoryReminder class can create new checkpoint when the originator class request. Data have been added to the HistoryReminder class with the state number. It can rollback or restore its previous state when a HistoryReminderOriginator request is done.

4.3.2. The HistoryReminderOriginator Class

The HistoryReminderOriginator class has an association with the HistoryReminder class. Hence, it is posible to create the HistoryReminder object as well as a checkpoint for the HistoryReminder class. It can rollback and return the data to the client class when requested.

When a client object initiates a transaction or ask for some service, it first requests the HistoryReminderOriginator object to créate a HistoryReminder object. The client object starts operations after creating its state. For creating a checkpoint object it requests the HistoryReminderOriginator object. After saving the data, the HistoryReminder object's returns to the client object with a new state. The client object continues its interactions with its corresponding HistoryReminder object. When a client needs to restore its previous state, it calls the rollback() operation. After that, the HistoryReminderOriginator object returns data with the current state. Thus, the "History Reminder" design pattern manages the state by storing and restoring the state.

The main ingredient of the "History Reminder" design pattern is that the client code will never access the HistoryReminder object or its operations. Thus, the HistoryReminder class can handle all checkpoints with the help of the HistoryReminderOriginator class and can restore the state with the help of the originator.

4.4. Applicability of the "History Reminder" design pattern

The "History Reminder" design pattern is used to save and restore the runtime state of an application. It holds the information by setting the History state at the runtime and restore when an operation needs to roll back. The below three situations emphasize the commercial value and application of the "History Reminder" design pattern:

- The banking system involves several simultaneous transactions. At any point of any transaction failure, it can rollback and restore to its previous state. The HistoryReminder object creates several checkpoints. Any transaction can go back and forward within the same object. In addition, data have been encapsulated and the transaction object was kept hidden from the outside classes.
- The database transaction and operation must have to be atomic, consistent, and durable. A transaction can have multiple operations and each operation is independent. If any of these transactions fails, then the entire system will crash. So all state changes will be committed together if all transactions succeeded, otherwise it will roll back to its previous state.
- To implement and design large scale of software application "HistoryReminder" design pattern is used to hold the state of each stage of the application. The

HistoryReminder object creates several check points to keep track of each state. It will restore the previous state when an application needs to rollback.

4.5. The Structure of the "History Reminder" design pattern

The "History Reminder" design pattern consists of two basic classes. The first one is the HistoryReminder class and the second one is HistoryReminderOriginator class. Here, we have removed the Caretaker class which has been used to restore and save the memento state in the case of the traditional well-known Memento design pattern. The Caretaker object's responsibility has been passed to the HistoryReminder object itself. It saves and restores the object within itself and returns it to the HistoryReminderOriginator object. The class diagram of the HistoryReminder design pattern is given below in Figure 3:

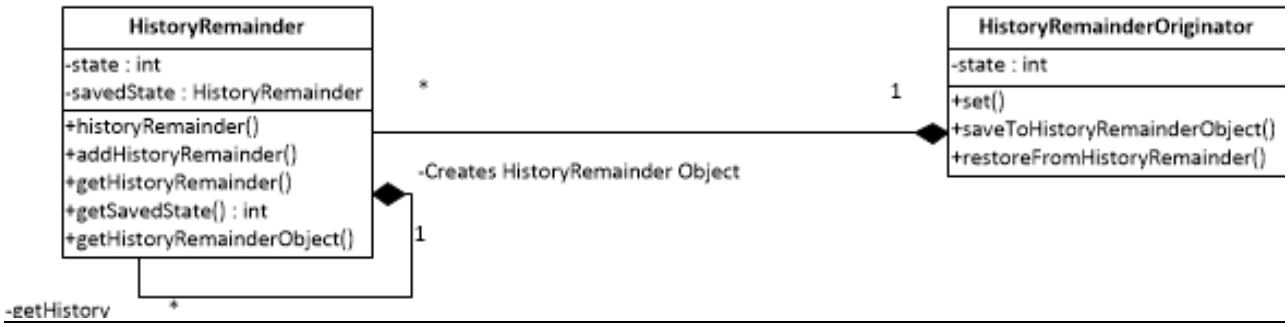

Figure 3. The class diagram of the "History Reminder" design pattern.

4.6. Collaborations of the "History Reminder" design pattern

The "History Reminder" design pattern has two main classes: HistoryReminder and HistoryReminderOriginator. The flow chart of the "History Reminder" design pattern is given in Figure 4.

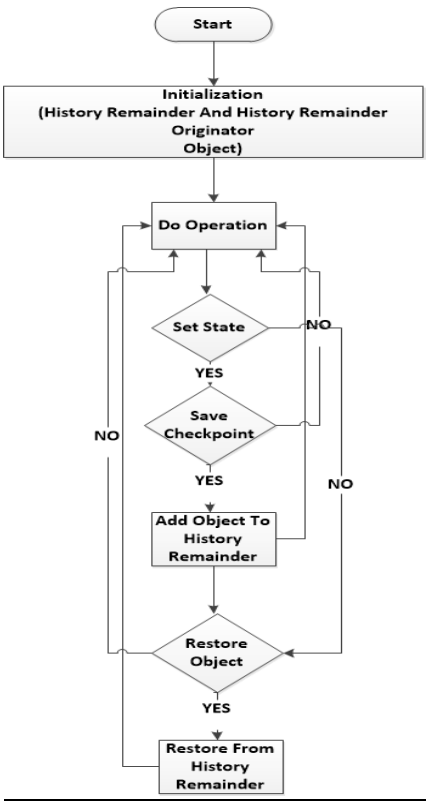

Figure 4. The Flow chart of the "History Reminder" design pattern

Analyzing the two main classes HistoryReminder and HistoryReminderOriginator of the "History Reminder" design pattern, we find that these two classes act as actors of the whole sequence of operations. The sequence diagram of the "History Reminder" design pattern is given below in Figure 5.

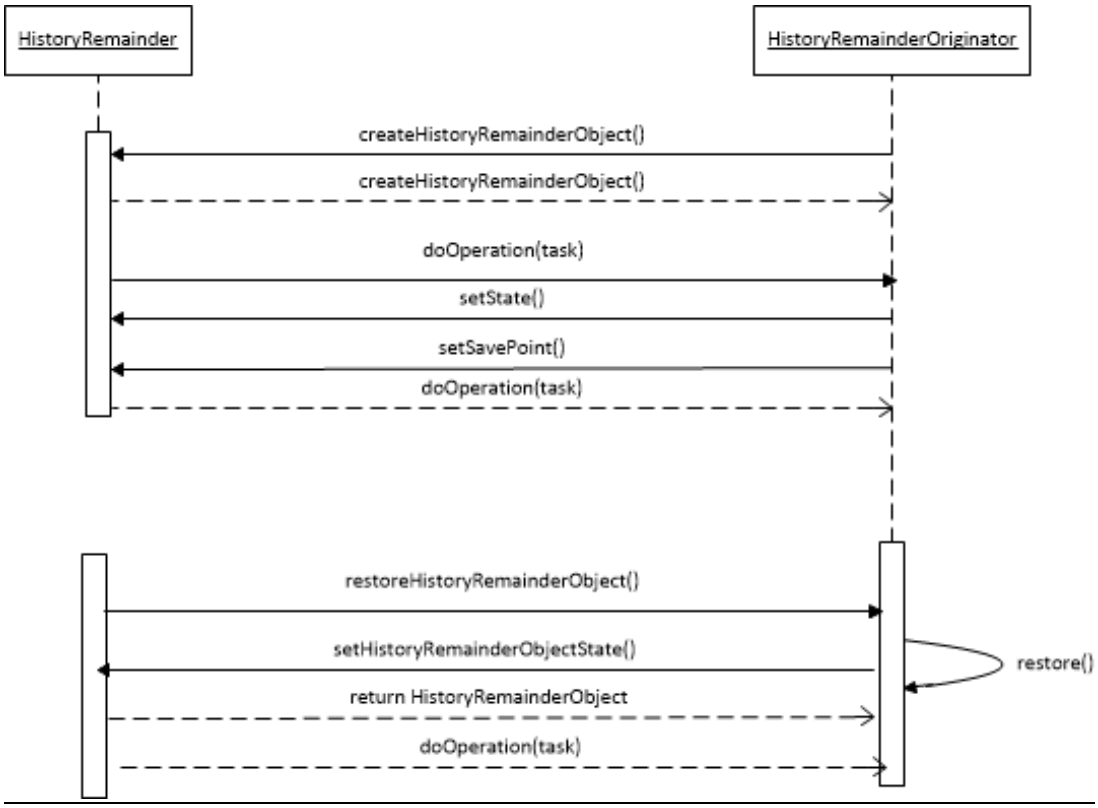

Figure 5. The Sequence Diagram of the "History Reminder" design pattern

4.7. Consequences of the "History Reminder" design pattern

The "History Reminder" design pattern is proposed to overcome the difficulties of Memento, State and several other design patterns. It has interconnections with other design patterns to design a large software application. However for a large scale of applications, we need to design a hybrid design pattern. Our proposed design pattern shows the existing relations with other design patterns. The "History Reminder" design pattern is a new pattern which can hold the history of transaction with its own object and can restore it when it is required. Here is the list of detailed improvements:

1. The "History Reminder" design pattern saves data based on the state variable of the HistoryReminder class. All the previous HistoryReminder objects' data have been stored in the array. If at any point we lose the information of the state or at any point we lose our state variable or number of states we will lose our entire previous objects. In case of Memento, there is no option to roll back to the previous state because of the lack of backup memory (in order to retrieve the object later). It may lead to a big problem when we will handle a large database system.

2. For any complex system, several clients access the same HistoryReminder object to store and restore data. If two clients request to create a checkpoint at the same time, then there may arise conflict to hold the current state. Then any of the client's request may not be executed and it will not create a checkpoint for that request.

3. The HistoryReminder class saves the HistoryReminder object. After that, if the HistoryReminderOriginator object cannot pass the current state to the client, then the

client cannot continue its operation which eventually conflict with the system. In addition, the operation will be invalid as the client does not know about the status of the previous state. For solving this problem we need to manage two flag variables for saving and restoring state.

4. The HistoryReminder class cannot be inherited from another class. Only the HistoryReminderOriginator class can access HistoryReminder class. It hides data from all classes except the originator. The HistoryReminderOriginator access the HistoryReminder class to create a new checkpoint. In case the HistoryReminderOriginator requests to creating new checkpoints without having a HistoryReminder object created, then it will lead a new conflict.

Beside these improvements, future features might also be considered in order to handle more complex software applications.

4.8. Implementation of the "History Reminder" design pattern

The HistoryReminder class is responsible for saving and restoring the HistoryReminder object. These classes perform all the operations corresponding to the Caretaker class and to the Memento class. A HistoryReminder object saves the information as a state variable and stores the states as a list of saved states. When a HistoryReminder object is created, it adds the HistoryReminder object to HistoryReminder class by using the function addHistoryReminder(). It restores the state and the information from getHistoryReminder() when the rollback() function is called. The getSavedState() function has been used to return the current saved state of the Memento object. The getHistoryReminderObject() is a static function which has been used to get the HistoryReminder object. Here we can pass the state of the HistoryReminder object as parameter. It checks the request of a valid HistoryReminderOriginator object. It also checks the requested state whether it is already being stored in the HistoryReminder object. Then it returns the HistoryReminder object of the requested state to the HistoryReminderOriginator object.

At a glance, the HistoryReminder class is responsible for getting the history of the previous state. The HistoryReminder class can have many HistoryReminder objects with different states. One HistoryReminderOriginator object can hold many HistoryReminder objects for restoring and saving purposes.

The HistoryReminder class of "History Reminder" design pattern has been given below in a Java-like style:

```
class HistoryReminder {
        int state;
        static ArrayList<HistoryReminder> savedStates =
                new ArrayList<HistoryReminder>();
        function static addHistoryReminder(HistoryReminder hr) {
                savedStates.add(hr);
        }
        function static HistoryReminder getHistoryReminder(int index) {
                return savedStates.get(index);
        }
        function HistoryReminder(intstateToSave) { state = stateToSave; }
        function int getSavedState() { return state; }
        function  static ArrayList<HistoryReminder> getHistoryReminderList() {
                return savedStates; }
        function  static HistoryReminder getHistoryReminderObject(int state, Object object) {
                if (object instanceof HistoryReminderOriginator)
                        return new HistoryReminder(state);
```

```
            else return null;
        }
} // end of Class HistoryReminder
```

The History Reminder Originator class is responsible for restoring and holding a HistoryReminder object. The client does not have a direct access to the HistoryReminder class and its object and properties. So it requests the HistoryReminder object to create and restore the states when required. The HistoryReminderOriginator class, along with the HistoryReminder class, hold the information and the state of the HistoryReminder object. It restores the information and returns back to the client when the rollback() operation is called. The state variable is used to recognize the current state of the HistoryReminder object. When a client requests to create checkpoint to a HistoryReminder originator, it calls the function sabe To History Reminder Object() and creates a checkpoint by adding a HistoryReminder object to the HistoryReminder class. When a client wants to restore and to rollback to its previous state, it calls the restore FromHistoryReminder() function. It restores and retrieves the information from the HistoryReminder class by using the getHistoryReminderObject() function and returns it to the client.

The HistoryReminderOriginator class of the "History Reminder" design pattern has been given below:

```
Class HistoryReminderOriginator {
        int state;
        function set(int setState) { state = setState; }

        function HistoryReminder saveToHistoryReminderHistoryReminderOriginator() {
                return HistoryReminder.getHistoryReminderObject(state, hro);
        }
        Function restoreFromHistoryReminder(HistoryReminderhr) {
                state = hr.getSavedState();
        }
} // end of Class HistoryReminderOriginator
```

The Main class of HistoryReminder design pattern is given below:

```
Class Main {
        HistoryReminder historyReminder = new HistoryReminder(1);
        HistoryReminderOriginator historyReminderOriginator =
                new HistoryReminderOriginator();
        HistoryReminderOriginator.set(2);
        HistoryReminder.addHistoryReminder
                (historyReminderOriginator.saveToHistoryReminder
                        (historyReminderOriginator));
        historyReminderOriginator.set(3);
        HistoryReminder.addHistoryReminder
                (historyReminderOriginator.saveToHistoryReminder
                        (historyReminderOriginator));

        historyReminderOriginator.restoreFromHistoryReminder
                (HistoryReminder.getHistoryReminder(1));
        HistoryReminder historyReminder2 = new HistoryReminder();
        historyReminderOriginator.set(8);
        HistoryReminder.addHistoryReminder
                (historyReminderOriginator.saveToHistoryReminder
```

```
                    (historyReminderOriginator));
        historyReminderOriginator.restoreFromHistoryReminder
                (HistoryReminder.getHistoryReminder(4));
} // end of Class Main
```

6. Conclusion and future work

In this paper, we have analyzed the existing software design patterns and described two new design patterns for saving the state of an object at runtime. Our "Obligation Link" and "History Reminder" design patterns may play an important role for large scale hybrid software application systems where saving and restoring objects at runtime are important to keep the software consistent. The "Obligation Link" and "History Reminder" design patterns can be used to reduce the execution time and some limitations of existing patterns. In addition, these two new design patterns can be used to improve complex systems.

References

Erich, G., Helm, R., Ralph, J., & Vlissides, J. (1997). *Design Patterns: Elements of Reusable object-oriented Software.* August 1997, Addison-Wesley.

Priestley, M. (2004). *Practical Object-Oriented Design with UML*, McGraw Hill

Larman, C. (2005). *Applying UML and Patterns*, Pearson – Prentice Hall

Wadhwa, B., Andrei, S, & Yuen Jien, S. (2007). *Software Engineering: An object-oriented approach*. McGraw Hill

IBM (1999). *IBM Rational Rose Enterprise*. [available at http://www-03.ibm.com/software/products/en/enterprise]

Tigris. (2005). ArgoUML. [available at http://argouml.tigris.org/]

Dougherty, C., Sayre, K., Seacord, R., Svoboda, D., & Togashi, K. (2009). *Secure Design Patterns.* CMU/SEI-2009-TR-010. Software Engineering Institute, Carnegie Mellon University. [available at http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9115]

Keutzer, K. & Mattson, T. (2010), *A Design Pattern Language for Engineering (Parallel) Software*. The Intel Technology Journal 13: 4.

Bashir, O. (2010). Using Design Patterns to Manage Complexity. *Overload Journal*, No. 96. April 2010 [available at: http://accu.org/index.php/journals/1622]

Knox, J. (2011). Adopting Software Design Patterns in an IT Organization: An Enterprise Approach to Add Operational Efficiencies and Strategic Benefits. *Technical Report of University of Oregon*.
[available at https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/11396/Knox-2011.pdf?sequence=1]

Noborikawa, M. (2003). Refactoring Software Using Design Patterns. *Technical Report of University of Northern Iowa*. May 5, 2003. Volume 39. [available at http://www.cs.uni.edu/~wallingf/miscellaneous/student-papers/noborikawa-paper.pdf]

OODesign. (2015). *Design Patterns*. [accessed on 10th May, 2015: http://www.oodesign.com/]

Pmsware. (2013). *Project Management and Software Engineering.* The Twenty Eleven Theme. January 14, 2013. [available at https://pmsware.wordpress.com/tag/computational-design-patterns/]

Kanasz, R. (2013). Design Patterns 3 of 3 - Behavioral Design Patterns. January 10, 2013. [available at http://www.codeproject.com/Articles/455228/Design-Patterns-of-Behavioral-Design-Patterns]

Sanders, B. (2014). *PHP Memento Design Pattern Part II: Store & Retrieve.* June 12, 2014. [available at http://www.php5dp.com/tag/php-memento-design-pattern/]

Shvets, A., Frey, G., Pavlova, M. (2010). Behavioral Patterns. *Sourcemaking*. [available at https://sourcemaking.com/design_patterns/behavioral_patterns]

Wei Wang, Hai Zhao, Hui Li, Peng Li, Dong Yao, Zheng Liu, Bo Li, Shuang Yu, Hong Liu, & Kunzhan Yang**.** (2010). Application of Design Patterns in Process **of** Large-Scale Software Evolving. Journal of Software Engineering and Applications (3): 58-64. [available at http://dx.doi.org/10.4236/jsea.2010.31007]

Pena-Mora, F. & Vadhavkar, S. (1996). Design Rationale and Design Patterns in Reusable Software Design. *Technical IESL Report No. 96-07*. Intelligent Engineering Systems Laboratory. Massachusetts Institute of Technology, U.S.A., November 1996

Dill, K., Pursel, E.R., Garrity, P., & Fragomeni, G. (2012). Design Patterns for the Configuration of Utility-Based AI. *Interservice/Industry Training, Simulation, and Education Conference* (*I/ITSEC*) 2012, No. 12146, pp. 1-12