ISSN No. 0976-5697

Volume 2, No. 2, Mar-Apr 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Optimization and Improving Security in Web Browsers Using the Proxy Pattern

Hassan Rashidi
Department of Statistics, Mathematics and Computer Science
Allameh Tabataba'i University
Tehran, Iran
hrashi@googlemail.com

Abstract: Design patterns are general reusable solutions to a commonly occurring problem in software design. One of the important patterns is Proxy Pattern. It is used for improving security in many systems such as separating an interface from a number of alternate implementations, wrapping around a set of legacy classes and so on. In this paper, a solution for display the content of tables in web pages by using the Proxy Pattern is proposed. This solution optimizes operations in browsers and improves security of system.

Keywords: design pattern, proxy pattern, security, browser, web page, table

I. INTRODUCTION

The idea of Design Patterns was introduced by Christopher Alexander, an architect, in the field of architecture. It has been then adapted for various other disciplines, including computer science [8]-[9]. In object-oriented approach, a design pattern is composed of a small number of classes that, through delegation and inheritance, provide a robust and modifiable solution. These classes can be adapted and refined for the specific system under construction. They are re-usable solutions to recurring problems. They were tried, tested and considered as solutions to be templates. Then they can be adapted and personalized for the problem domain. In addition, design patterns provide examples of inheritance and delegation.

Many patterns have been proposed for a broad variety of problems in software engineering [1], including analysis [Fowler, 1997] [Larman, 1998], system design [Buschmann et al., 1996], middleware [Mowbray & Malveau, 1997], process modeling [Ambler, 1998], dependency management [Feiler et al., 1998], and configuration management [Brown et al., 1999]. Buschmann et al [3] categorize patterns into three groups, Architectural Patterns, Design Patterns, and Idioms. Architectural Patterns are very high-level structure for software systems and contain a set of predefined sub-systems. They define the responsibilities of each sub-system and detail the relationships between sub-systems. Design Patterns are mid-level construct and Implementation-independent. They are designed for 'micro-architectures' and somewhere between sub-system and individual components. Idioms are earliest form of software pattern. They are comparatively low-level and give a guide for implementing the components and relationships of the pattern. Idioms consider the pattern at a programming language level and describe the patterns using the constructs of the specific language.

This paper focuses on Design Patterns and presents a solution to improve security in browsers and optimize operations by using the Proxy Pattern when they display the contents of a table in web pages. The structure of the remaining sections is as follows: Section 2 reviews design patterns and makes a classification of them. Section 3 explains Proxy pattern and gives the applications of this pattern. Section 4 shows how the Proxy Pattern improves security in web browsers when a page contains some tables. Section 5 is considered for conclusion and future research.

II. DESIGN PATTERNS

Using patterns to define a software solution is an analytical task that requires abstract thinking. Design patterns are template designs in form of several classes in object-oriented design. They are methods of encapsulating the knowledge of experienced software designers in a human readable and understandable form. They are particularly appropriate in situations where classes are likely to be reused in a system that evolves over time.

Design patterns are partial solutions to common problems that can be used in a variety of systems such as separating an interface from a number of alternate implementations, wrapping around a set of legacy classes, protecting a caller from changes associated with specific platforms [1]. Design Patterns make object-oriented software more reusable, flexible, modular and understandable.

Each Design Pattern is associated with a Name, Description, Solution and Consequences. Some of the names used by Gamma et al[2] have become standard software terminology. Description of each Pattern describes when it might be used, often in terms of modifiability and extensibility. Solution is expressed in terms of classes and interfaces and consequences are related to trade-offs and alternatives.

Table 1 classifies Design Patterns into three groups: *creational*, *structural* and *behavioral* and presents the most important patterns in each group [4]. The creational design patterns are all about class instantiation. These patterns can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done. The structural design patterns are all about Class and Object composition. The structural class-creation patterns use inheritance to compose interfaces. The structural

object-patterns define ways to compose objects to obtain new functionality. The behavioral design patterns are all about Class's objects communication. These patterns are those patterns that are most specifically concerned with communication between objects.

Table I: A Classification of Design Patterns

Groups	Major Patterns	Description
Creational patterns	Abstract Factory	Creates an instance of several families of classes
	Builder	Separates object construction from its representation
	Factory Method	Creates an instance of several derived classes
	Object Pool	Avoids expensive acquisition and releases resources by recycling objects that are no longer in use
	Prototype	A fully initialized instance to be copied or cloned
	Singleton	A class of which only a single instance can exist
Structural patterns	Adapter	Matches interfaces of different classes
	Bridge	Separates an object's interface from its implementation
	Composite	A tree structure of simple and composite objects
	Decorator	Adds responsibilities to objects dynamically
	Facade	A single class that represents an entire subsystem
	Flyweight	A fine-grained instance used for efficient sharing
	Private Class Data	Restricts accessor/mutator access
	Proxy	An object representing another object
Behavioral patterns	Chain of responsibility	A way of passing a request between a chain of objects
	Command	Encapsulates a command request as an object
	Interpreter	A way to include language elements in a program
	Iterator	Sequentially accesses the elements of a collection
	Mediator	Defines simplified communication between classes
	Memento	Captures and restores an object's internal state
	Null Object	Designed to act as a default value of an object
	Observer	Notifies changes to a number of classes
	State	Alters an object's behavior when its state changes
	Strategy	Encapsulates an algorithm inside a class
	Template method	Defers the exact steps of an algorithm to subclass
	Visitor	Defines a new operation to a class without change

III. PROXY PATTERN

This section focuses on Proxy Pattern and shows how it improves the performance or the security of a system by delaying expensive computations, using memory only when needed or checking access before loading an object into memory.

In UML (Unified Modeling Language), classes are depicted as boxes with three sections, the top one indicates the name of the class, the middle one lists the attributes of the class, and the third one lists the methods [5]-[7]. A typical UML Class diagram for an object filename in operating system is shown in Fig. 1 by using Proxy Pattern. The ***ProxyObject*** class acts on behalf of a ***RealObject*** class. Both classes implement the same interface. The *ProxyObject* stores a subset of the attributes of the *RealObject*. The *ProxyObject* handles certain requests completely (e.g., determining the size of an image), whereas others are delegates to the *RealObject*. After delegation, the *RealObject* is created and loaded in memory. The consequence of using Proxy patterns is that they add a level of indirection between Client and RealObject. The Client is shielded from any optimization for creating RealObject.

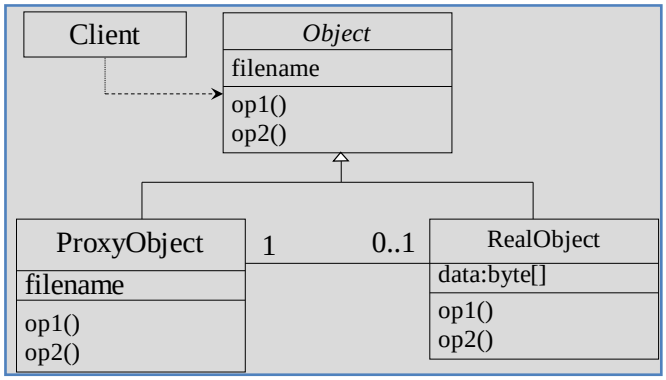

Figure 1: A typical representation of Object filename using Proxy Pattern (UML Class Diagram)

The Proxy Patterns are used for Protection. Fig. 2 shows an example for Protection with four classes: ***Broker, Access, PortfolioProxy*** and ***Proxy***. The Access association class contains a set of operations that a Broker can use to access a Portfolio. Every operation in the PortfolioProxy first checks with *isAccessible*() if the invoking Broker has legitimate access. Once access has been granted, PortfolioProxy delegates the operation to the actual Portfolio object. If access is denied, the actual Portfolio object is not loaded into memory. One Access association can be used to control access to many Portfolios.

The Proxy Patterns are also used for Storage. One example is illustrated in Fig. 3 in which an image class (left side) is converted to three classes (right side). An ***ImageProxy*** object acts on behalf of an Image stored on disk. The ImageProxy contains the same information as the Image (e.g., width, height, position, resolution) except for the Image contents. The ImageProxy services all contents independent requests. Only when the Image contents need to be accessed (e.g., when it is drawn on the screen), the ImageProxy creates the RealImage object and loads its contents from disk.

Moreover, caching expensive computations is another application of Proxy Patterns. In this application, expensive computations often only need to be done once, because the base values from which the computation is done do not change or change slowly. In such cases, the result of the computation can be cached as a private attribute. Consider, for example the ***Layer.getOutline***() operation in Fig. 4, as part of a Geographical Information Subsystem (GIS) [1]. All ***LayerElements*** are defined once as part of the configuration of the system and do not change during the execution. Then, the vector of Points returned by the Layer.getOutline () operation is always the same for a given ***bbox*** and detail. Moreover, end users have the tendency to focus on a limited number of points around the map as they focus on a specific city or region. Taking into account these observations, a simple optimization is to add

a private ***cachedPoints*** attribute to the Layer class, which remembers the result of the getOutline () operation for given bbox and detail pairs. The getOutline() operation then checks the cachedPoints attribute first, returns the corresponding Point Vector, if found, it otherwise invokes the getOutline () operation on each ***containedLayerElement***. Note that this approach includes a trade-off: On the one hand, we improve the average response time for the getOutline () operation; on the other hand, we consume memory space by storing redundant information.

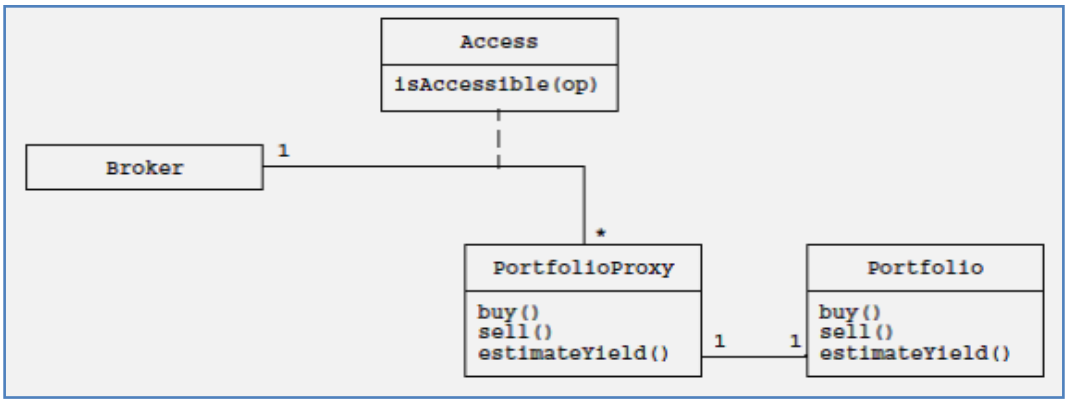

Figure 2: A UML class diagram for dynamic access implemented with a protection using Proxy Pattern.

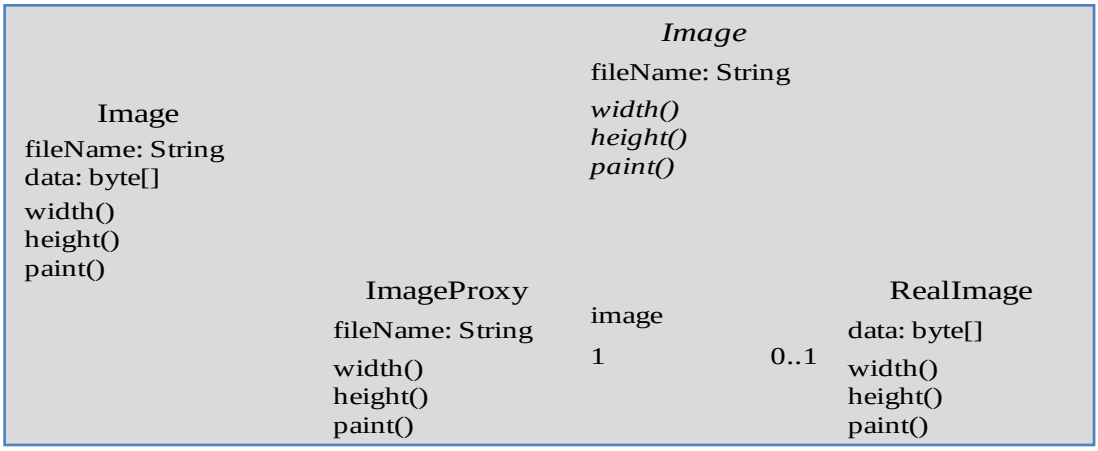

Figure 3: A UML class diagram for storage using Proxy Pattern.

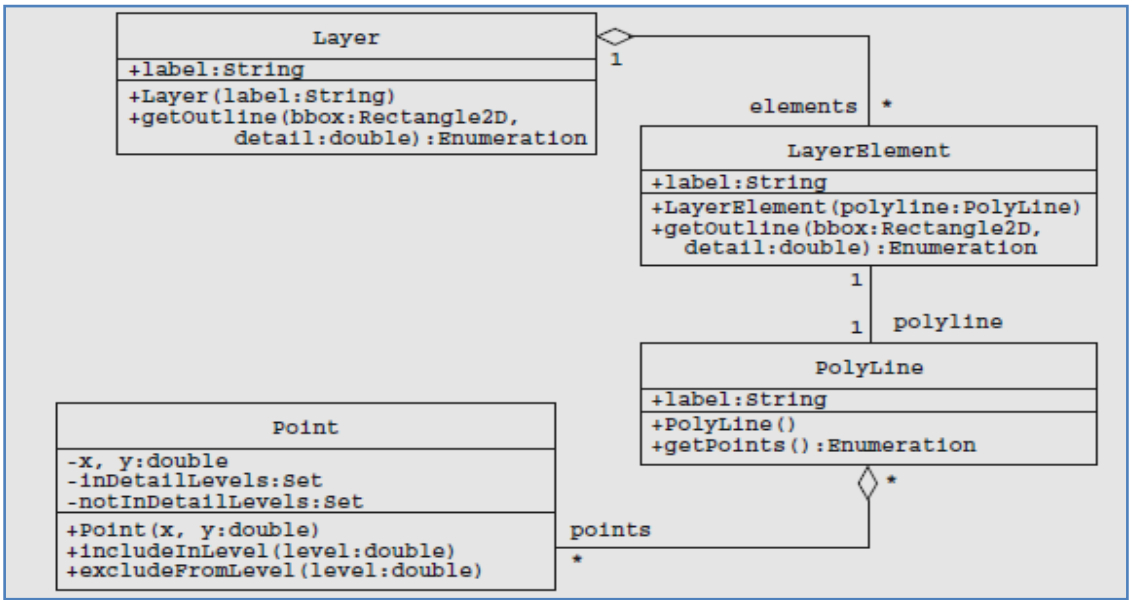

Figure 4: A UML class diagram for caching expensive computation using Proxy Pattern, adding type information to the object model of the GIS

IV. USING THE PROXY PATTERN IN BROSWERS

There are several goals when a designer wants to design an Object in Object oriented approach. One important goal is to optimize objects and delaying expensive computations as long as possible. Another goal is to improve security. This section presents a solution method when we want to optimize operations and improve security in web page in which has some tables.

Consider an object representing an image stored as a file. Loading all the pixels that constitute the image from the file is expensive. However, the image data does not need to be loaded until the image is displayed. We can realize such an optimization using a ***Proxy pattern*** [2]. An ***ImageProxy*** object takes the place of the Image and provides the same interface as the Image object, as shown in Fig. 3. Simple operations such as *width* () and *height* () are handled by *ImageProxy*. When Image needs to be drawn, however, ImageProxy loads the data from disk and creates a ***RealImage*** object. If the client does not invokes the *paint* () operation, the RealImage object is not created, thus saving substantial computation time. The calling classes only access the ImageProxy and the RealImage through the Image interface.

Web pages usually contain of several tables. The tables themselves consist of rows, which in turn consist of cells. The actual width and height of each cell is computed based in part on its content (e.g., the amount of text in the cell, the size of an image in the cell), and the height of a row is the maximum of the heights of all cells in the row. Consequently, the final layout of a table in a Web page can only be computed once the content of each cell has been retrieved from the Internet.

Using the proxy pattern described in Fig. 3, we can present an object model and a solution that enables a Web browser to start displaying a table before the size of all cells is known, possibly redrawing the table as the content of each cell is downloaded. The solution is illustrated in the UML class diagram of Fig. 5. There are five classes in the diagram: ***Table, TableRow, TableCell, TableCellProxy*** and ***RealTableCell***. In the table shown, there are four attributes (x, y, w, h) and the method *paint* (). The attributes are considered for position, width, and height of the table. The *Table* is composed of several *TableRows* and each *TableRow* is composed of several *TableCells*. When the table is created, ***TableCellProxy*** estimates its dimensions as best as it can update them as the content is retrieved. Every time *TableCellProxy* provides a new estimate of its dimensions, it signals *Table* via a semaphore or a listener [6]. The *Table* then recomputes the dimensions and positions of all of its rows and redraws itself.

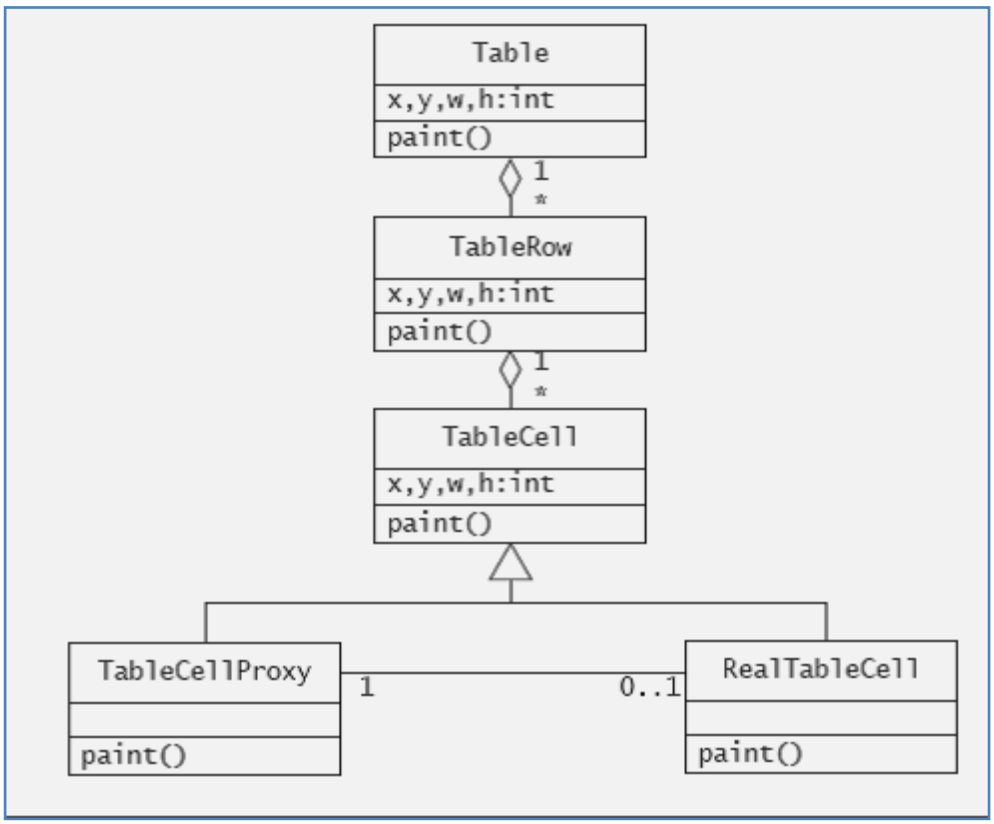

Figure 5: The class diagram for the solution.

V. CONCLUSION AND FUTURE WORK

Design patterns are methods that enable reuse of code and good solutions since it contains experiences from successful solutions to other similar problems. One of the important design patterns is Proxy Pattern of which is used when we need to represent a complex object with a simpler one. If creation of object is much expensive, its creation can be postponed till the very need arises and then, a simple object can represent it.

This paper proposed a solution for display contents of tables in web pages by using the Proxy Patterns. This solution can be used in browsers that improve security of systems and optimize the design. Implementing of the solution, finding a good estimator for dimension of tables

and collecting results from experiments is proposed for future research.

VI. REFERENCES

[1]. Bruegge B. and Dutoit A.H, "Object-Oriented Software Engineering: Using UML, Patterns, and Java", Prentice Hall, 2004.

[2]. Gamma E., Helm R., Johnson R., and Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[3]. Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M., "Pattern-Oriented Software Architecture: A System of Patterns", Wiley, Chichester, U.K., 1996.

[4]. 101 Design Patterns & Tips for Developers, available on web at http://sourcemaking.com/design_patterns.

[5]. Larman C., "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design", 3rd Edition, Prentice Hall, 2004.

[6]. Schmidt D., Stal M., Rohnert H. and Buschmann F., "Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects", Volume 2, John Wiley & Sons, 2000.

[7]. Booch G., Rumbaugh J, and Jacobson I, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1999.

[8]. Wolfgang P., Hermann S., "Design Patterns-Essentials, Experience, Java Case Study", Fourth Asia-Pacific Software Engineering and International Computer Science Conference, pp. 534-535, 1997.

[9]. Wolfgang P., Hermann S., "Design Patterns for Object-Oriented Software Development (Tutorial)", Proceedings of the 19th international conference on Software engineering, pp. 663-664, 1997.