J Inf Process Syst, Vol.9, No.4, pp.567-574, December 2013
http://dx.doi.org/10.3745/JIPS.2013.9.4.567
pISSN 1976-913X
eISSN 2092-805X

The Architectural Pattern of a Highly Extensible System for the Asynchronous Processing of a Large Amount of Data

Ro Man Hwang*, Soo Kyun Kim*, Syungog An*, and Dong-Won Park*

Abstract—In this paper, we have proposed an architectural solution for a system for the visualization and modification of large amounts of data. The pattern is based on an asynchronous execution of programmable commands and a reflective approach of an object structure composition. The described pattern provides great flexibility, which helps adopting it easily to custom application needs. We have implemented a system based on the described pattern. The implemented system presents an innovative approach for a dynamic data object initialization and a flexible system for asynchronous interaction with data sources. We believe that this system can help software developers increase the quality and the production speed of their software products.

Keywords—Large data, UML Diagram, Object-Oriented Software

1. INTRODUCTION

These days the amount of information required by human hands is constantly increasing. In turn, results in an increase of a demand on software applications that allow representing, analyzing, and modifying this information in a more convenient way. The sources of this amount of information could be a local database, a corporate server database, or even the Internet.

In spite of the similarities of such applications, it is almost impossible to create a single software framework for all of the applications, because every application domain has its own specifics and rules for data processing. A developer, who wants to create a single solution, risks receiving a product that is overloaded with a bulk of settings that must be tuned for a specific problem. Moreover, this kind of solution is difficult, and therefore, expensive, to implement and support.

The best approach in this situation is the creation of a single solution—a pattern for the construction of such applications. By following this pattern, developers would be able to effectively create products to solve specific problems. The expenses of the implementation and integration to a custom software product will be slightly decreased.

2. REQUIREMENT ANALYSIS

The main requirement for an architectural pattern [1] is abstraction. The pattern should be ap-

Manuscript received January 21, 2013; accepted March 8, 2013.
Corresponding Author: Dong-Won Park (dwpark@pcu.ac.kr)
* Dept. of Game Engineering, Paichai University, Deajeon, 302-735, Korea (minlog@empas.com, nicesk@gmail.com, dwpark@pcu.ac.kr, sungohk@pcu.ac.kr)

plicable to a large range of problems. The next section describes the common problems of such software applications.

2.1 Cascading the propagation of lower level modifications

Changes in a data source structure result in serious changes in all the application layers. Even when using a three-tier architecture approach [2] (See Fig. 1), the changes often cause a cascading modification of each layer up to the user interface (the Presentation layer). Some might note that a careful upfront design of a data source structure can eliminate such problems. However, such a design is virtually impossible in reality. The advent of iterative software development methodologies, such as SCRUM [3, 4] and Extreme Programming [5], proves this point.

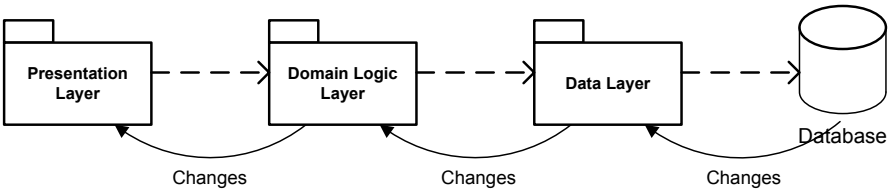


Fig. 1. The propagation of changes from the data source to the Presentation Layer in three-layer architecture [2]

2.2 The conflict between a relational database and an object-oriented system

It is much more convenient for a programmer or a user to work with data that is represented in an object-oriented way, when there is no need to care about the characteristics of relational databases, such as an implementation of the "many-to-many" relation. This kind of relation requires the creation of an additional link table [7], which does not have any representation in an object-oriented paradigm. This relationship is modeled by having each object contain an array of other objects (See Fig. 2).

However, in an object-oriented way, this relation can be easily described with the following code in C# (See Fig. 3):

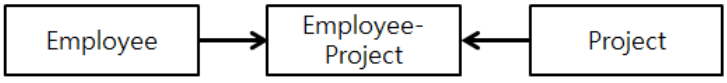


Fig. 2. A many-to-many relation as represented in a relational database

```
class Employee
{
    public List<Project> projects;
}

class Project
{
    Public List<Employee> employees;
}
```

Fig. 3. Example of a many-to-many relationship represented using the C# code

2.3 Executing long running operation results in a user interface freeze

If you are using a single threaded application with a graphical user interface, every time you are accessing a data source the user interface will freeze until the access is finished. Even working with high-speed local databases does not guarantee that freezing will not occur. During the synchronous execution of a complicated query, a graphical user interface becomes irresponsible.

Based on the problems described above, we will present an outline below about the requirements for the system:

1. Support of an asynchronous execution of queries.
2. Execute a scalable number of execution threads.
3. Have the ability to set a synchronous execution mode for a debugging purpose.
4. Changes in a data source must not affect more than one architectural layer.
5. Be independent of the number and type of data sources.

The next section contains the description of a system that satisfies the above-listed requirements.

3. REFLECTIVE DATA REPRESENTATION

The system has adopted a graph model for representing data (See Fig. 4). Thus, every entity from an application domain can both contain a collection of child entities and be a child of other entities. This gives us tremendous power in representing virtually any complex structure.

We define an artifact, as an object that represents some entity from an application domain. Basically, the artifact has a name, a list of properties, and a list of values associated with the properties. Because an artifact is just a collection of properties, it does not exactly represent an entity in an application domain, but it can represent any entity in a generalized way. This could be seen as a negative performance hit to the system. However, the purpose of the system is to flexibly manage large amounts of data with slow data sources. Thus, gaining (or increasing) flexibility is much more important than decreasing the number of execution cycles.

In order to enclose a structure of the artifact from its representation in a database, we used a reflective approach [8, 9]. Reflection is a mechanism of meta-programming that allows obtaining

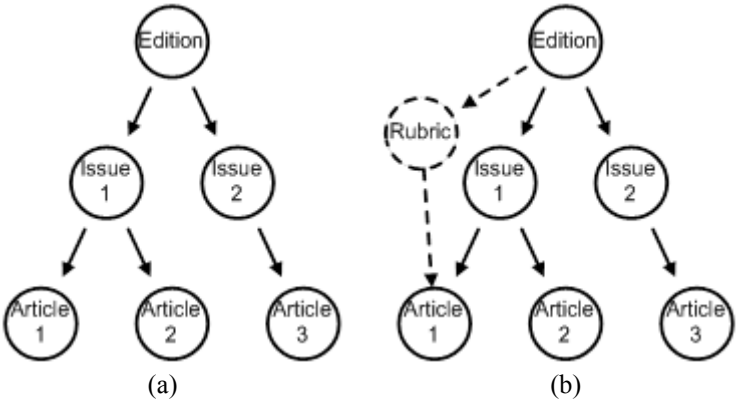


Fig. 4. Example of tree (a) and graph model (b) for a data representation of a publishing house database

information about a class structure during the run-time [9]. The described system has a dynamic reflection of an artifact structure in a database on the artifact class. The number of properties and their value is defined during the creation of each artifact, and depends on the information returned from a database.

Fig. 5 provides a UML diagram of the artifacts of a publishing house database.

By our design each artifact contains an associative array, which is also known as a dictionary or a map, where the key is a property name, and the value is a property value that is retrieved

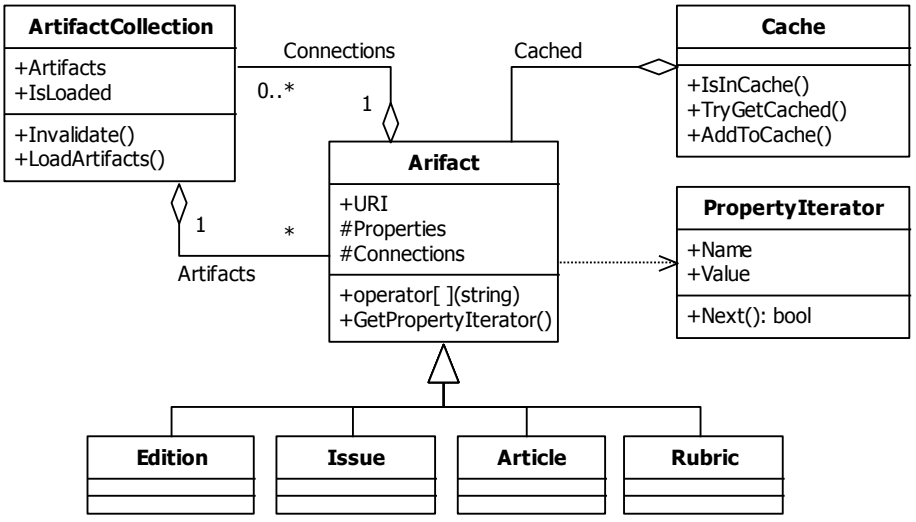


Fig. 5. UML class diagram of artifacts of a publishing house database

```
class ControlFactory
{
    IControl CreateControl(string name, object value)
{
    switch(name)
{
    case “PublicationDate”: return new DateControl(name, value);
    case “Hyperlink”: return new HyperLinkControl(name, value);
    default: return new LabelControl(name, value);
}
}
}

foreach (var item in artifact.Properties)
{
    var control = controlFactory.CreateControl(item.Name, item.Value);
        gui.AppendControl(control);
}
```

from data source. This approach helps to significantly protect the data and the domain logic layers from the actual structure of an artifact. The only thing that is depending on the real artifact structure is the presentation layer and data modification algorithms. However, both the presentation layer and the data modification algorithms can dynamically adapt to changes to the structure.

A C#-based source code sample below shows how a representation layer can dynamically adopt to changes in the data structure. A programmer can flexibly extend the controlFactory to generate different controls depending on property name, and can provide default control for items that did not match a specific property name:

To make an abstraction of the artifact structure even greater we have introduced property iterator [1] classes. Iterators can greatly help to create architecture with low coupling. For example, in the case when an application is required to visualize a table with the names and properties of an artifact, the presentation layer does not need to know the name of each in order to be able to retrieve it. It just needs to iterate all of the properties and to visualize a property name as the table row, and a property value as the table cell value. The sample code above demonstrates the usage of iterator patter in C#. However, the pattern is widely used in many other programming languages, such as Java, Python, C++. For a C++ standard template library (STL), the pattern is made up of well-known key components together with collections and algorithms, and it acts as a connection mechanism between them [10, 11].

4. The System of Asynchronous Interaction With Data Sources

This section provides a description about the system of asynchronous interaction. The main class of the system, which is called "Core," is responsible for the initialization of all the subsystems and the interaction with the presentation layer (GUI) through the IArtifactView interface (See Fig. 6). An object that implements the interface and registers itself in the Core, can receive a notification each time the data in modified. The design of using an interface for view, without any direct references from the Core object to the actual IArtifactView implementer class, is based on the Model-View-Controller (MVC) architectural pattern [12].

During the initialization phase, the Core class constructs the necessary object of class that implements the ICommandQueue interface. Classes from this hierarchy implement a command queue. The command, which is put into the queue, will be executed asynchronously in a background thread.

Thus, during the initialization of one of the classes that implements IArtifaceView, a class can query the loading of required artifacts. This query will be processed asynchronously and by the end of it all of the classes that implement the IArtifaceView interface will be notified about the update of data (See Fig. 7). This approach helps to eliminate the freezing of a graphical user interface during the query processing, and can even visualize the process of loading, if a certain mechanism is supported.

The command class significantly extends the capabilities of the queue. A command can encapsulate any action that requires a long execution time. The Core class mechanisms notify all the IArtifactView classes about the execution process and the completion of the execution.

Most of the graphical user interface building systems (MFC, .Net Windows Forms) limit access to their graphical objects only by the main thread. This forces the implementation of a dispatch system, which moves the notification about the finishing of an executed command to the main

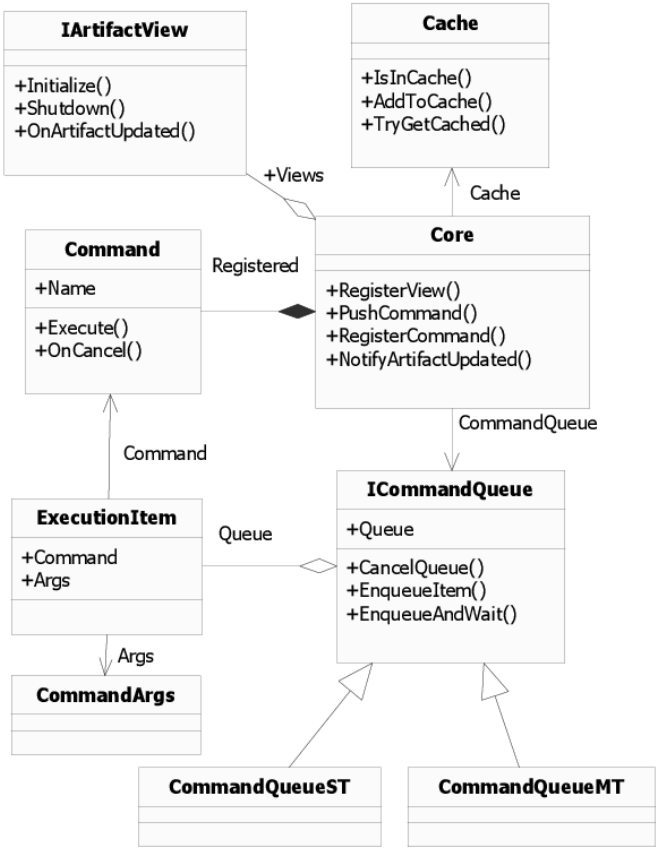


Fig. 6. Diagram of core classes and interfaces of the system

graphical user interface thread.

In the described implementation there is only one instance of each command, and this instance is put to the queue using ExecutionItem objects, together with all that is necessary for the execution arguments. Because one command can be simultaneously executed in several threads, it is implemented as an object without a state (stateless) and can only work with data passed as parameters.

The artifact cache (Cache class) is designed to secure the uniqueness between an artifact inside a data source and an artifact in the program. We have introduced the URI (Unique Resource Identifier) for the unique identification of artifacts. If a currently loading artifact is already in the cache, the loading is cancelled and a reference to the existing artifact is returned.

Besides asynchronous execution benefits, the described model proposes a wide spectrum of modifications. On the basis of the existing system, it is possible to implement the following items: a command and parameters parsing for the command line execution; an implementation of the execution of commands in an arbitrary number of command threads (in the case where the Internet is the data source, it will provide a huge performance boost); and the support of an unlimited number of data visualization objects that are synchronized both with the database and each other.

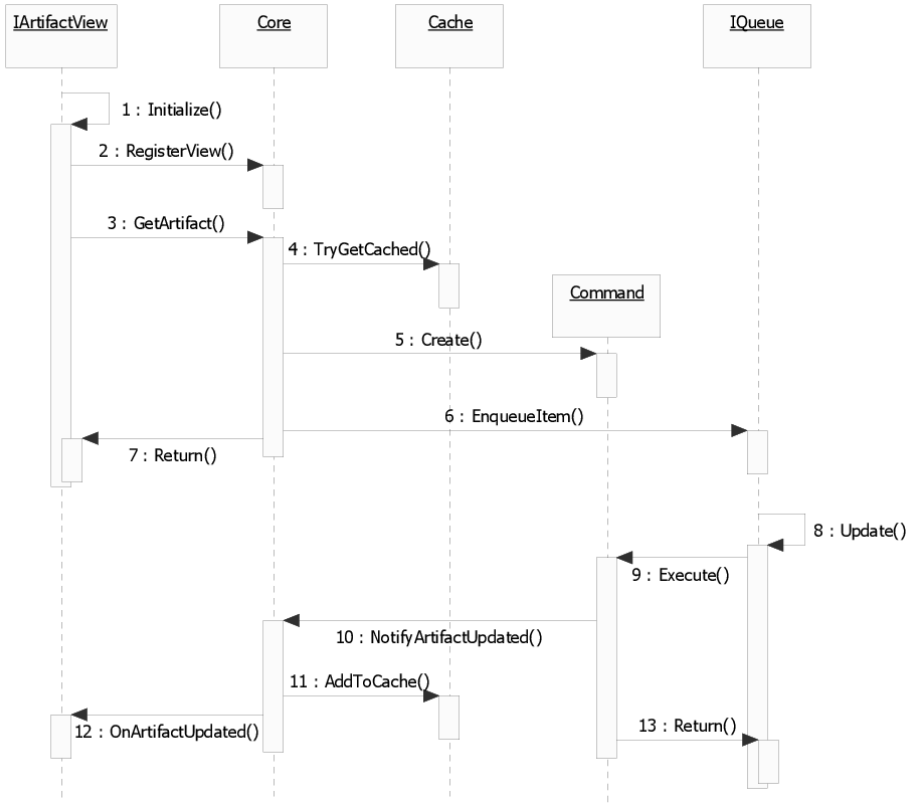


Fig. 7. Sequence diagram of IArtifactView initialization and an artifact update

5. CONCLUSION

This work presented an architectural pattern for applications that interact with low-speed data sources. The described pattern provides great flexibility, which helps in easily adopting it to a custom application needs. We have implemented a system based on the described pattern. The implemented system presents an innovative approach for the initialization of a dynamic data object and a flexible system for an asynchronous interaction with data sources.

The approach used in the described system is highly flexible and is easily extensible for interaction with application domain logic and presentation layers. We believe that this system can help software developers increase the quality and the production speed of their software products.

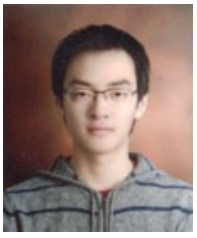
REFERENCES

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

[2] Martin Fauler, *Patterns of Enterprise Application Architecture*, 2002. Addison-Wesley Professional.

[3] Ken Schwaber, Mike Beedle, *Agile Software Development with Scrum*, 2001. Prentice Hall.

[4] Ken Schwaber, *Agile Project Management with Scrum*, 2004. Microsoft Press.
[5] Kent Beck, Cynthia Andres, *Extreme Programming Explained: Embrace Change*, 2004. Addison-Wesley Professional; 2ed.
[6] Bjarne Stroustrup, *The C++ Programming Language: Special Edition*, 2000. Addison-Wesley Professional.
[7] Avi Silberschatz, Hank F. Korth, S. Sudarshan, *Database System Concepts*, 2006. McGraw-Hill.
[8] Arthur H. Lee, Joseph L. Zachary,*" Reflections on Metaprogramming,"* November 1995. IEEE Transactions on Software Engineering, Volume 21, Number 11.
[9] Manuel Clavel, *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*, 2000. Cambridge University Press.
[10] Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2012. Addison-Wesley Professional; 2ed.
[11] Scott Meyers, *50 Specific Ways to Improve Your Use of the Standard Template Library*, 2001. Addison-Wesley Professional; 1ed.
[12] Erik M. Buck, Donald A. Yackman, *Cocoa Design Patterns*, 2009. Addison-Wesley Professional; 1ed.



Ro Man Hwang
He received the MS Degree in Game & Multimedia Eng. from Paichai University in 2012.

Soo Kyun Kim
He received Ph.D. in Department of Computer Science & Engineering from Korea University in 2006. He joined Telecommunication R&D center at Samsung Electronics Co., Ltd., from 2006 and 2008. He is now a professor at Department of Game Engineering at Paichai University, Korea. His research interests include multimedia, pattern recognition, image processing, mobile graphics, geometric modeling, and interactive computer graphics.

Syungog An
She received the Ph.D. in Computer Science from Korea University in 1989. She has been a professor at PAICHAI University since 1991. Her research interests include Computer Graphics, DataBase and Game Development.

Dong-Won Park
He received the Ph.D. in Computer Science from Texas A&M University in 1993. He has been a professor at PAICHAI University since 1994. His research interests include Networked Multimedia and Embedded S/W.