

The Configuration Data Caching Pattern

Leon Welicki
ONO (Cableuropa S.A.U.)
Basauri 7-9, 28027
Aravaca, Madrid, Spain
+34 91 180 950
lwelicki@acm.org

ABSTRACT

The CONFIGURATION DATA CACHING pattern describes how an application can consume efficiently configuration data stored in external repositories (databases, files, etc.). This is achieved by avoiding reacquisition of the parameters, holding the retrieved parameters in memory in a cache.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Miscellaneous – *reusable software*, D.3.2 Language Classifications – *Very high-level languages*, H.3.m Information Storage and Retrieval – *miscellaneous*

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Languages.

Keywords

Caching, Patterns, Design Patterns, Performance, Configuration Parameters.

1. CONTEXT

A software system has configuration information stored in an external secondary-memory based repository, so details are not hard-coded in code. The configuration information is accessed during the program's execution.

Configuration parameters may change sporadically, so the cache should be updated (manually by an administrator or automatically by a synchronization mechanism) to keep the information in the cache up to date.

2. EXAMPLE

Consider a web based mortgage simulator for buying houses. When the user enters the application she is presented with a screen where she has to introduce her birth date, annual income, amount of money needed and years of mortgage. Using all this data as input the system makes its calculation and gives the user

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLoP '06, October 21–23, 2006, Portland, OR, USA

Copyright 2006 ACM 978-1-60558-151-4/06/10...\$5.00.

an average monthly cost.

The calculation has some special rules: if the solicitant is younger than a certain age she gets a special discount (a help given by the government to young people for buying their first home). If she has a low income she gets another special government help. The system also checks that the average monthly cost of the mortgage don't exceed a certain percentage of the solicitant's income (if that percentage is exceeded the mortgage is denied).

The values used by the system to perform the calculation are fixed for relatively long periods of time but may change according to the markets and government regulations, so having them "hard-coded" in the source code of the simulator is a very bad idea. Therefore, these values are stored in a configuration file, having entries for the interest rate, the maximum age in which a person can apply for the government help for young people, the threshold for determining low income, and the maximum allowed percentage of the income that can be dedicated to pay the mortgage.

Every time a mortgage is calculated the application needs to read the configuration file, parse it, and get all the configuration parameters quoted above. This adds an overhead to the task of performing the calculation, since every time it is performed the configuration file should be read and parsed. This overhead may be significant and may affect the scalability (and as a side effect the availability) of the application, since as more requests for calculations the application receives, more readings and parsing of the configuration file must be done (in this case, our mortgage simulator application can "die of success").

3. PROBLEM

How can we consume efficiently configuration parameters stored in a persistent secondary storage repository so they have not to be reloaded every time they are requested?

4. FORCES

- *Flexibility.* The configuration parameters of an application should not be hard-coded in the application.
- *Performance.* The cost of repetitious configuration parameters acquisition must be minimized. The configuration parameters are often stored in slow secondary memory devices such as hard drives (which are slower than memory).
- *Scalability.* The solution should be scalable regarding the number of configuration parameters and concurrent users.
- *Availability.* The solution should allow cached configuration information to be accessible even when the repositories are temporarily unavailable. Additionally, it should not be

necessary restarting the consumer application to clear the configuration parameters cache.

- *Simplicity*. The solution should be easy to use. It does not have to add unnecessary complexity to the application where is included.
- *Control*. The solution should not make the program state opaque. The contents of the configuration parameter cache should be accessible by an administrator to determine the causes of the behavior of the application.

5. SOLUTION

Store the value of the parameter in memory when it is requested for the first time. Use the value stored in memory in the next requests for the same parameter, so data has not to be fetched again from the configuration repository. Provide a mechanism to clear outdated (or invalid) parameters from memory.

When the application needs to use a configuration parameter, it requests it to the configuration data cache. If the configuration data cache contains the parameter, it returns the configuration parameter instance that it has stored. If the configuration data cache does not contain the parameter, it connects to the configuration data repository (which can be a file, a database, etc.), fetches the parameter, buffers it and returns it to the application. Subsequently, when that parameter is requested again the cache returns the buffered value.

An application administrator must be able to clear the cache whenever he desires. Additionally, the cache may have some synchronization or eviction mechanism as presented in the CACHING pattern in [4] (or in the variants section later in this pattern) that would perform the synchronization process automatically.

Optionally, administrators should be able to browse the contents of the cache and delete specific elements (which will allow him to know the contents of the cache and therefore determine the cause of a program behavior).

6. STRUCTURE

The following participants form the structure of the CONFIGURATION DATA CACHING pattern:

- A *config param* is a configuration parameter, such as a connection string to the database or the address of a server. It may contain metadata (such as date added or hit count) to assist the synchronization and eviction processes.
- A *config param user* uses configuration parameters (config param).
- A *config param cache* buffers configuration parameters. Once a parameter is stored in the configuration parameter cache it is not fetched again from the repository.
- A *config param provider* fetches data from the persistent configuration parameters repository and returns an instance of a config param.

The CRC cards in figure 1 show how the participants interact with each other.

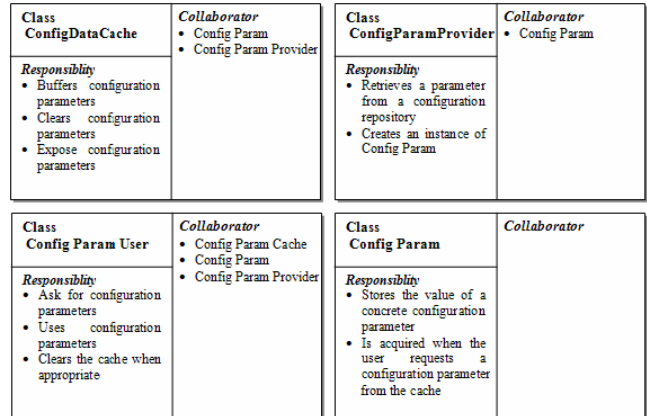


Figure 1. CRC Cards for the participants of the solution

The UML class diagram in figure 2 illustrates the structure of the Configuration Data Caching Pattern.

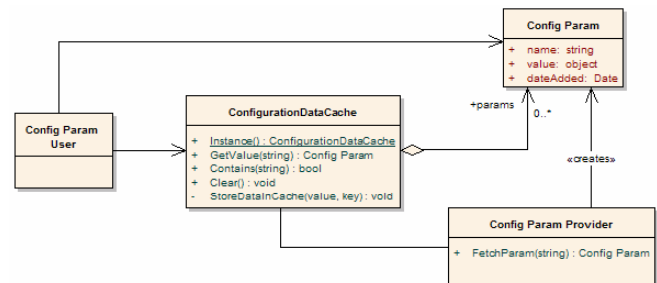


Figure 2. UML Class Diagram of the solution

7. DYNAMICS

The figure 3 shows how the participants interact to get a new configuration parameter. In this case, the parameter is not stored in the cache and therefore it is fetched from a persistent configuration repository.

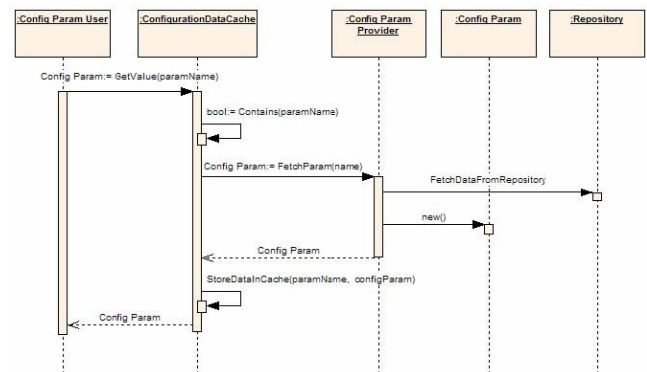


Figure 3. Dynamics for retrieving a new parameter

The figure 4 shows how the participants interact when a configuration parameter that is stored in the cache is requested.

Notice that in this last case, the ConfigParamProvider is not used because the parameter is already stored in the parameters cache and is not necessary to fetch it from the persistent repository neither create the parameter instance.

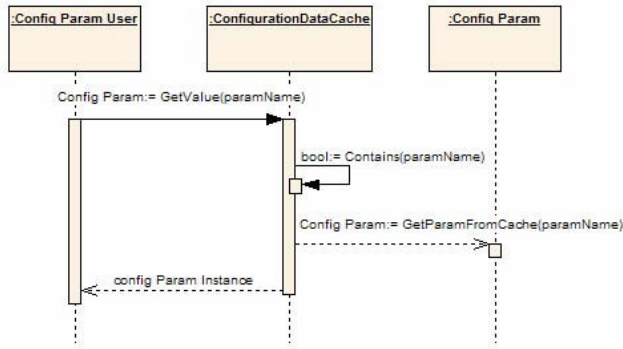


Figure 4. Dynamics for retrieving a cached parameter

The dynamics in the CONFIGURATION DATA CACHING pattern are simpler than in the CACHING pattern ([4], pp. 87): the cached resources are always requested to the cache whether the request is the first or the hundredth. The user does not need to be aware of the caching status of a particular resource. This simplifies dramatically the usage of the cache at expenses of some flexibility that the CACHING pattern offers. Since one of our forces is simplicity, enhance in the ease of usage outweighs the loss of flexibility.

8. IMPLEMENTATION

To implement the CONFIGURATION DATA CACHING pattern, the following steps should be followed:

1. *Select the configuration parameter type.* Select the type of the configuration parameters that is going to be managed by the cache. The configuration parameter may have descriptive metadata in order to (added date, hit count, etc.) that may be used by the eviction or synchronization mechanism.

2. *Select the configuration source.* Select the configuration information data source. The configuration data source is where the configurations parameters are stored persistently. The information source can be a configuration file, a table in a database, etc.

Important note on steps 1 and 2: when an existing application is retrofitted to use a cache this two steps are already decided (according to the application’s actual architecture).

3. *Create the configuration data cache.* The configuration data cache is the responsible of giving the parameters to the user. The dynamics diagram in the previous section shows how this participant handles user requests, stores the parameters in memory and when a parameter is not found asks it to the data provider.

4. *Create the provider to fetch data.* The provider is the responsible of the communication with the actual persistent repository where the data lives. The provider could be designed using the STRATEGY [2] pattern so different data access strategies may be used within the same cache according to specific needs.

5. *(optional) Add an eviction strategy.* An eviction strategy may be used to evict invalid or outdated parameters in the cache. The EVICTOR [4] may need the metadata associated with the cached parameter. For example, we may have an eviction strategy that removes from the cache all the values that are older than one hour or another that removes the unused parameters. The eviction

strategy adds complexity to our solution, but does not affect the user of the cache, since the usage interface of the cache is designed for simplicity. The complexity lies in the internals of the cache and is not exposed to end users.

6. *(optional) Add synchronization.* The parameter cache may be synchronized with the parameters repository so whenever a parameter changes the cache is automatically updated. A discussion on this issue can be found in the CACHING pattern at [4].

7. *Using the configuration parameters cache.* The user of the configuration cache parameters uses the cache. As we stated above, using the cache must be very easy. The end-user must not be aware if the parameter is being retrieved from the original source of information (configuration file, database, etc.) or from the cache.

9. CONSEQUENCES

The configuration data cache adds performance and flexibility to an application. The access to configuration data is improved, since data is fetched only once. Since having all the necessary configuration information does not hurt performance anymore it supports enhancing an application’s flexibility (moving fixed values to configuration).

This helps to put in practice the principle “put abstractions in code and details in metadata” [3] (in this case metadata refers to configuration information), but reducing the performance overhead caused by retrieving the data stored in a secondary memory based repository.

9.1 Benefits

There are several benefits of using this pattern:

- *Performance.* Fast access to frequently used resources is an explicit benefit of CACHING [4]. In this case, we have fast access to the configuration parameters, since once they are loaded they remain cached in memory.
- *Flexibility.* We can bring more flexibility to our application, since we have an infrastructure that helps to manage the configuration parameters and performance penalty regarding using configuration data parameters is minimized.
- *Configurability.* Using configuration parameters is not a performance bottleneck anymore. All the variable parameters can be stored in a configuration repository. The access to this repository will be managed by the configuration data cache.
- *Availability.* Once the parameters are stored in the cache, the original source is not needed until the cache is cleared or a new parameter needs to be loaded. Reboots may not be needed to refresh the parameters in the in-memory cache.
- *Scalability.* Access to secondary-memory configuration for retrieving parameters is minimized; reducing the overhead and avoiding degradation proportional to the amount of concurrent users. This also avoids problems like dead-locks in configuration files when concurrent users try to access to the same parameter at the same time.
- *Better use of resources.* The implementation of the parameters cache makes a better use of resources. Data is only fetched when appropriate and then is stored in a fast access memory structure, which provides a very efficient access to it.

9.2 Liabilities

There are several liabilities of using this pattern:

- *Complexity*. The use of caching techniques adds complexity to an application.
- *Synchronization*. The parameters in the cache may be outdated or incorrect. Keeping the cache synchronized with the repository is a complex issue that when addressed adds more complexity to the overall solution.
- *Possible “over-effort”*. When an entry in the cache is used only once, it is wasting space in the cache and it has wasted time to put it in the cache. There is nothing that can be done against the waste of time when the entry is stored in the cache, but the memory usage can be fixed having an eviction strategy [4] that harvests the unnecessary information stored in the cache.

10. SAMPLE CODE

In the sample code included next, our cache class is implemented using a SINGLETON [2] with DOUBLE-CHECK LOCK [5] that hosts a Hashtable [6] with the configuration parameters instances. The hashtable represents a collection of key/value pairs that are organized based on the hash code of the key. It provides a simple and efficient mechanism for fast access to the cached configuration parameters (based on a dictionary with a constant lookup time). It can also safely support one writer and multiple readers concurrently using a built-in synchronization mechanism [6].

The modifications to the Hashtable are synchronized using locks to make the cache thread-safe (this can be observed in the StoreDataInCache method). Whenever a user requests a configuration parameter to the cache (using the GetValue method), if it is in memory it is given back to the user. If not, it is retrieved from the repository.

In the case of this sample, and for the sake of simplicity, we will just use strings for holding cached configuration parameter without any descriptive metadata (added date, etc.).

The next code snippet shows how a sample implementation of the configuration data cache class (in C#):

```
public class ConfigurationDataCache {
    private static ConfigurationDataCache
        instance = null;

    private static readonly object
        padlock = new object();

    private Hashtable data = null;

    private ConfigurationDataCache() {
        data = new Hashtable();
    }

    public static ConfigurationDataCache Instance {
        get {
            if (instance == null) {
                lock(padlock) {
```

```
                    if (instance == null)
                        instance =
                            new ConfigurationDataCache();
                }
            }
            return instance;
        }
    }

    private void StoreDataInCache(string key,
        string val) {
        lock (instance.data.SyncRoot) {
            if (instance.data.ContainsKey(key))
                instance.data.Remove(key);
            instance.data.Add(key, val);
        }
    }

    public string GetValue(string key) {
        string ret = null;

        if (instance.data.ContainsKey(key)) {
            ret = instance.data[key].ToString();
        }
        else {
            ConfigDataProvider provider =
                new ConfigDataProvider();
            ret = provider.GetParam(key);
            if (ret != null)
                this.StoreDataInCache(key, ret);
        }
        return ret;
    }

    public void Clear() {
        lock(instance.data.SyncRoot) {
            instance.data.Clear();
        }
    }
}
```

Notice that when a value is not contained in the cache, it is fetched from the repository by the ConfigDataProvider. The cache does not have any knowledge on how to retrieve the information from the data source. Another very important aspect of the class shown above is the method Clear, which allows clearing the contents in the cache deterministically in runtime (which would allow reloading the parameters stored in the cache without a process restart).

The next code snippet shows the implementation of a configuration data provider that fetches the configuration parameters from a .NET XML configuration file (through the AppSettings collection of the .NET framework's class ConfigurationSettings).

The `ConfigDataProvider` instance is initialized when the `ConfigDataCache` instance is created (see `ConfigDataCache` code above). The provider could be designed using the Strategy [2] pattern so different data access strategies may be used within the same cache according to specific needs. In this case the `ConfigDataCache` can be initialized (via dependency injection in the constructor) with a `ConfigDataProvider` strategy.

The next code snippet shows a sample usage of the cache created above. In this sample, a url and a port are fetched from the configuration repository (through the cache) to compose a connection string to a service. Notice that the code for consuming the configuration parameters does not make any distinction whether the parameters come from the configuration source (file, database, etc.) or the in-memory cache.

```
string urlParam =
    ConfigurationDataCache.
        Instance.GetValue("url");

string portParam =
    ConfigurationDataCache.
        Instance.GetValue("port");

string connectionString = urlParam + portParam;
```

11. EXAMPLE RESOLVED

A configuration data cache is used to manage the parameters. When each parameter is first requested, it is stored on the configuration data cache. Therefore, in the next requests for a parameter is not necessary to fetch it from the configuration data file. This avoids unnecessary reads and parses to the configuration file, since the calculation parameters are kept in memory by the configuration data cache.

If any parameter needs to be changed (according to market trends or governmental regulations) the administrator can modify the configuration file and the clear the contents of the cache. In the successive requests the process explained in the first paragraph of the solution is repeated, refreshing the contents stored in the cache. This avoids a process restart to reflect the changes in the configuration of the system, keeping the application always up to date and running.

12. VARIANTS

12.1 Synchronized Configuration Data Caching

The synchronized data cache has a synchronization mechanism that keeps the data in the cache synchronized with the data in the configuration repository. Synchronization is a very complex issue when dealing with these caches, since it may add a performance overhead that might overcome the benefits of using the cache. Therefore it must be designed carefully, balancing the trade-off of accuracy against performance. There are several ways to handle synchronization in a configuration data cache:

- *Manual Synchronization*: an administrator can trigger manually a cache reload or synchronization (the `Clear` method in the sample code above is an example of this technique). There are several ways to do this: if the cache is an eager cache (explained in the next variant), the cache is completely reloaded

on demand. If is a regular configuration data cache it just can be cleared and parameters are going to be reloaded as they are requested.

- *Periodic Rollup*: the cached parameter is reloaded periodically after a fixed number of reads. The possibility of using outdated parameters is $1/n$ (where n is the number of reads until the cache is refreshed). Periodicity can be measured also by time (in this case the parameters stored in the cache may have an associated timestamp that would determine their validity).
- *Polling*: there is a process that polls for changes in the source of information. When a parameter changes, the cache is informed and consequently updated. An OBSERVER [2] can be used to inform the cache about modifications in the source of information. ASP.NET [6] uses a similar approach to keep configuration up to date (it “listens” for changes to the `web.config` file).
- *“Manual-Automatic” Synchronization*: this method combines ideas from all the previous ones. Changes to the configuration parameters must be done through a user interface provided by the application. When a parameter is changed the interface invokes the cache synchronization (which can be based on removing the changed parameter and waiting for a new request of it, reloading the parameter, clearing all the cache, triggering the listener, etc.).

12.2 Eager Configuration Data Caching

When a program starts the cache retrieves all configuration values and stores them in memory. This eliminates the overhead associated with opening, scanning and parsing the configuration file each time a parameter is requested for the first time. Once the program has started and the cache has been loaded, the cost of using parameters is minimal. This may be better used when a big amount of the parameters are going to be used every time the program is run. The main liabilities of this variant is that it can make program start slower and that memory footprint of the cache may be bigger. The eager configuration data cache should be synchronized or should provide a “reload” mechanism (triggered by an administrator) to avoid application reboot when a parameter changes.

13. KNOWN USES

Microsoft ASP.NET configuration parameters from the `web.config` file are stored in memory after they are read. When a parameter changes they are synchronized again (but at the expense of restarting the process, which has the drawback of dropping non-persistent sessions).

Operating Systems (Windows, UNIX, Linux, etc.) store configuration parameters in memory. In the case of Windows, there is not a good synchronization strategy, nor cache clearance mechanism. Therefore, when some parameters change a reboot is necessary to apply the changes.

Databases often keep service configuration parameters in memory. For example, in Oracle databases (version 9 and below) the service information is stored in a configuration file called `init<SID>.ora` (where `SID` is the service identifier of the database). Once the instance is started the parameters are loaded in a view in memory (`V$PARAMETER`). There is no automatic synchronization between the file and the in-memory cache. To

apply changes in the configuration file a reboot of the service is required (which must be done by a database administrator).

Struts configuration file (*struts-config.xml*) is loaded into memory when the host application is started (this is similar to the EAGER CONFIGURATION DATA CACHING variant explained in the Variants section).

14. RELATED PATTERNS

The CONFIGURATION DATA CACHING is a variant of the CACHING pattern [4] that refines and extends it in the context of using configuration data in an application. The management of the cached resources and overall implementation is more complex and general-purpose in the CACHING pattern.

An EVICTOR [4] may be used for eviction of configuration data stored in the cache. For example, never accessed or old values may be evicted periodically.

The ConfigurationDataCache participant is often a SINGLETON [2].

STRATEGY [2] may be used to change the provider strategy (ConfigDataProvider) to fetch data (a provider may have several strategies aimed to fetch data from different kinds of repository, e.g., XML, relational database, flat file, etc.)

FACTORY METHOD [2] can be used to control the way in which ConfigParams instances are created in the provider.

15. ACKNOWLEDGMENTS

Thanks to Paul Adamczyk, the shepherd of this paper. I could never have written this pattern without his outstanding shepherding work. Thanks also to Kyle Brown, the program committee member during the shepherding process.

This paper benefited from the caring and the feedback of the members of PLoP 2006 Writer's Workshop - Ralph Johnson, Paddy Fagan, Jason Yip, Hesham Saadawi, Pau Arumi, Djamel Bellebia, Dirk Riehle, David Garcia) – led by Ralph Johnson. Thanks to all of them for their great feedback and insight.

16. REFERENCES

- [1] Buschmann, Frank et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996
- [2] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995
- [3] Hunt, Andrew; David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000.
- [4] Kircher, Michael; Prashant Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.
- [5] Schmidt Douglas; Tim Harrison. *Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects* in *Pattern Languages of Program Design* (Martin, Buschmann, and Riehle, eds.), Addison Wesley. 1997.
- [6] Microsoft .NET Home Page.
<http://www.microsoft.com/net/default.aspx>