The Configuration Data Caching Pattern

Leon Welicki
lwelicki@acm.org

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.

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.

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 that she needs and years of mortgage. Using all this data as input the system makes its calculation and gives the user 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 houses). If she has a low income she gets also another special government help. The system also checks that the average monthly cost of the mortgage don't exceed a percentage of the solicitant's income (if the percentage is exceeded the mortgage is denied).

The values used by system to perform the calculation are fixed for relatively long periods of time, but 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 a considerable 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 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 simulation application can "die of success").

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?

Forces

- *Flexibility*. The configuration parameters of an application should not be hard-coded in the application, because they can change.
- *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 ostensible slower than memory).
- *Scalability*. The solution should be scalable regarding the number of configuration parameters and concurrent users.
- *Availability*. The solution should allow the cached configuration information to be accessible even when the repositories are temporarily unavailable. Additionally, it should not be necessary to restart 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.

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.

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 data base, 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 [POSA3].

Optionally, administrators should be able to browse the contents of the cache and delete specific elements.

This helps to put in practice the principle "put abstractions in code and details in metadata" [HT00] (in this case metadata refers to configuration information), by reducing the performance overhead caused by retrieving the data stored in a secondary memory based repository.

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.
- 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 following CRC cards show how the participants interact with each other:

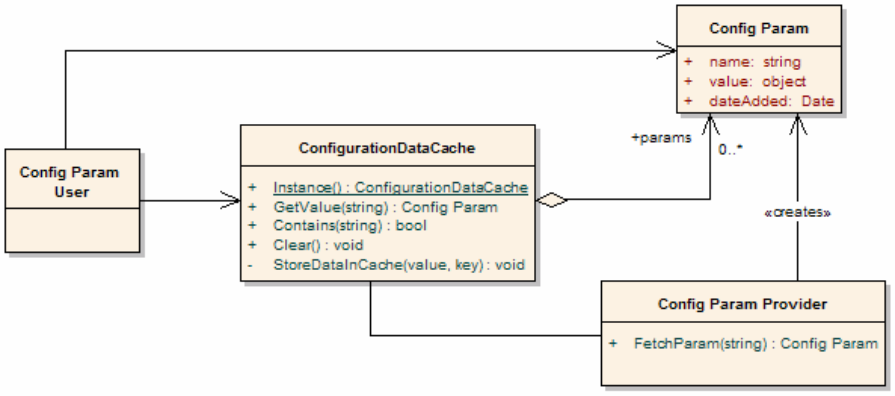
Class **ConfigDataCache**	*Collaborator* • Config Param • Config Param Provider
Responsiblity • Buffers configuration parameters • Clears configuration parameters • Expose configuration parameters	

Class **ConfigParamProvider**	*Collaborator* • Config Param
Responsiblity • Retrieves a parameter from a configuration repository • Creates an instance of Config Param	

Class **Config Param User**	*Collaborator* • Config Param Cache • Config Param • Config Param Provider
Responsiblity • Ask for configuration parameters • Uses configuration parameters • Clears the cache when appropriate	

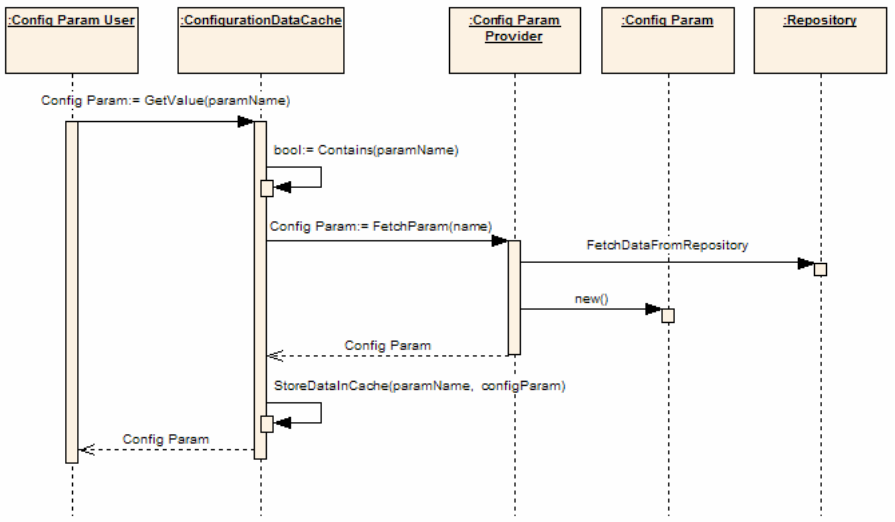
Class **Config Param**	*Collaborator*
Responsiblity • Stores the value of a concrete configuration parameter • Is acquired when the user requests a configuration parameter from the cache	

The following class diagram illustrates the structure of the Caching pattern.

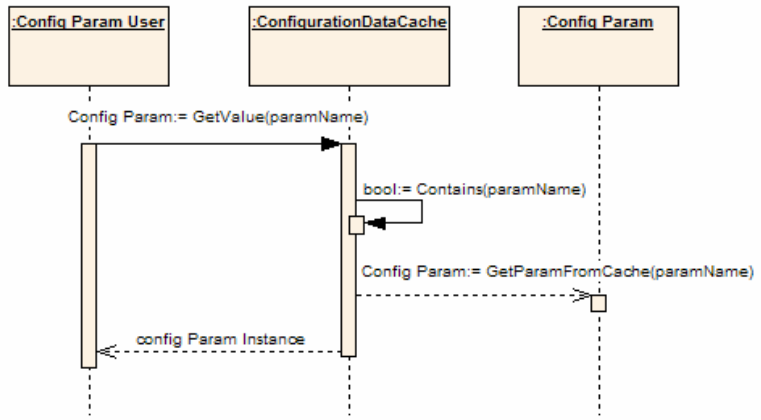


Dynamics

The following figure 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.



The next figure shows how the participants interact when a configuration parameter that is stored in the cache is requested. Notice that in this last case, the `Config Param Provider` 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.



The dynamics in the CONFIGURATION DATA CACHE pattern are simpler than in the CACHING pattern ([POSA3], pp 87): the cached resources are always requested to the cache whether the first or the hundredth request is. 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 losing some flexibility that the CACHING pattern offers. Since one of our forces is *simplicity*, enhance eased of use outweighs the loss of flexibility.

Implementation To implement the CONFIGURATION DATA CACHE 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. In the case of this sample, for the sake of simplicity, we will just use strings and we are not going to add descriptive metadata (added date, etc.) to the cached configuration parameter.

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 data base, 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.

   In the sample code included next, our cache class is implemented using a SINGLETON [GoF95] with DOUBLE-CHECK LOCK [SHR97] that hosts a `Hashtable` [@NET] 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 [MSDN].

   The modifications to the `Hashtable` are synchronized using `locks` to make the cache thread-safe (this can be observed in the `StoreDataInCache` method). As we stated in the step 1, our cache will manage string values 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. 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 ConfigDataProvider provider = null;

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

   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 {
         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. We will address the data provider in the next step.

Another very important aspect of the class shown above is the method `Clear`, which allows clearing the contents in the cache deterministically at runtime. Invoking `instance.data.Clear()` is better than doing `instance.data = null`, because removes all entries from the current instance and lets the instance ready for storing new elements. Setting the instance to null is worst for two reasons: 1) it lets the instance unusable for storing new elements and 2) makes a more inefficient use of memory (the instance and all its data is marked for being garbage collected).

Finally, there is a very important side effect of having the parameters in this in-memory cache: if the source of information becomes unavailable it won't affect our application availability since we are not fetching the parameters from the source anymore but using the ones that we already have in memory.

Create the provider to fetch data. The provider is the responsible of the communication with the actual persistent repository where the data lives. The next code snippet shows the implementation of the provider that fetches the configuration parameters from the .NET XML configuration file (through the `AppSettings` collection of the .NET framework's class `ConfigurationSettings`).

```
public class ConfigDataProvider
{
   public string FetchParam(string name) {
      return ConfigurationSettings.AppSettings[name];
   }
}
```

In the sample code in step 3 the `ConfigDataProvider` is instance initialized when the `ConfigDataCache` instance is created.

The provider could be designed using the STRATEGY [GoF95] 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.

4. *(optional) Add an eviction strategy*. An eviction strategy may be used to evict invalid or outdated parameters in the cache.

The EVICTOR [POSA3] 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.

Adding eviction to the cache requires having metadata associated with each cached value to determine when an item must be evicted. For example having the date when the element was added allows evicting the older elements; having the number of times an item is accessed helps to evict less used items.

5. *(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 [POSA3]. The SYNCHRONIZED CONFIGURATION DATA CACHING variant (presented later in the variants section) discusses this issue in more depth.

6. *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 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:

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

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

string connectionString = urlParam + portParam;
```

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).

There are several **benefits** of this pattern:

- *Performance*. Fast access to frequently used resources is an

explicit benefit of caching [POSA3]. In this pattern, 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 are not 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.

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 also 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

that harvests the unnecessary information stored in the cache.

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 the any parameter needs to be changed (according to market trends or government regulations) the administrator can modify the configuration file and then 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.

Variants

- **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 file. 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 the configuration data cache:

    - *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)

    - *Listener*: there is a listener process that checks for changes in the source of information. When a parameter changes, the cache is informed by the listener and consequently updated. An OBSERVER [GoF95] can be used to inform the cache the modifications in the source of information. ASP.NET [ASPNET] uses a similar approach to keep configuration up to date (it "listens" for changes to the `web.config` file).

    - *Manual Synchronization*: an administrator can trigger manually a cache reload or synchronization. There are several ways to do this: if the cache is an eager cache (explained in the next bullet), the cache is completely reloaded on demand. If is a regular configuration data

cache it just can be cleared and the parameters are going to be reloaded as they are requested.

- *"Manual-Automatic" Synchronization*: this method combines ideas from all the previous. 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, reloading the parameter, clearing all the cache, triggering the active listener observers, etc.)

- **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 used 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 change.

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 the 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.** 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.** 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 explained in the variants section).

Related Patterns

The CONFIGURATION DATA CACHING is a variant of the CACHING pattern [POSA3] 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 [POSA3] may be used for eviction of configuration data stored in the cache. For example, never accessed or old values may be evicted periodically.

The CONFIGURATION DATA CACHE participant is often a SINGLETON [GoF95]

STRATEGY [GoF95] may be used to change the provider strategy 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 [GoF95] can be used to control the way in which `Config Params` instances are created in the provider.

References

[GoF95] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.

[HT00] Hunt, Andrew; David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000.

[POSA1] Buschman, Frank et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996.

[POSA3] Kircher, Michael; Prashant Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[SH97] Schmidt Douglas; Harrison Tim. *Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects*. http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf

[NET] Microsoft .NET Home Page. http://www.microsoft.com/net/default.mspx

[ASPNET] Microsoft ASP.NET Home Page. http://www.asp.net

[MSDN] Microsoft Developers Network: Visual C# Developer Center. http://msdn.microsoft.com/vcsharp/