Configuration Provider:
A Pattern for Configuring Threaded Applications

Klaus Meffert[1] and Ilka Philippow[2]
Technical University Ilmenau
plop@klaus-meffert.de[1], ilka.philippow@tu-ilmenau.de[2]

Abstract

You have an application that reads in configuration data at start-up. Your program consists of multiple threads with similar functionality. Some of the configuration data applies to all threads; the other part of the data applies to individual threads only. The configuration data should be read-only after it has been provided at start-up. This paper introduces the pattern *Configuration Provider* as an implementation suggestion for this context.

Keywords: design patterns, application configuration, threads, Java.

1 Context

1.1 Thumbnail

The following categories display a brief perspective for the suitable environment of the pattern.

Systems: General purpose systems (such as personal computers or workstations)
Applications: Multi-threaded object-oriented applications; configurable applications and threads
Users: Application developers, software architects
Languages: Java, possibly any object-oriented language similar to Java (e.g. C#)

1.2 Use Case

The use case includes multi-threaded applications that rely on global as well as on thread-specific read-only configurations.

1.3 Setup

A threaded application owns two different configuration types. The first one is the thread-specific configuration that is unique for each thread. The second one is the global application configuration. Each thread should have access to both configurations. Furthermore, each configuration object should be immutable after its initialization is finished. The reason for relying on threads in the paper is that there may be threads with similar functionality and thus similar configuration requirements living in the same application.

2 Considerations

The objects that build your application need access to application-specific configuration data. A common solution for solving this problem would be to use a Singleton [2] to provide the configuration data application-wide. This solution could also be applied to a threaded class.

Now: the configuration data must be available to each thread after start-up. Obviously it may lead to buggy application states in case a thread accesses the configuration before it is actually read in completely. When using a Singleton you could solve the problem of reading an uninitialized or partly initialized configuration. For this the Singleton is required to provide some locking mechanism which releases the lock after the configuration has been determined. Relying on a static variable that stores the configuration is not feasible here as locking could not be provided easily. A lot of discussions and resources exist that indicate the difficulties with implementing a locked access in a Singleton.

The process of reading the configuration may be time-consuming. You therefore have to ensure that it is not executed more than once. Even in case the reading process is not critical in time, it may not be a good idea executing it twice because between two read accesses to a configuration file, e.g., the contents of the file could have changed. Implementing a mechanism to avoid doubled processes is also possible via a lock. But it should be considered that a thread accessing the configuration could be constrained by a timeout. In case calling the Singleton returning the configuration when a lock is set because the reading process is ongoing, the timeout constraint may come into play. It could be avoided running a thread before the configuration is fully available, but this meant for the developer knowing about this problem and to invest manual effort and energy in solving it. Singleton itself does not seem to offer a self-contained solution to this problem. For the problem of preventing processing configuration resources twice, a double-checked locking mechanism might be better to use than the simple lock from the previous figure.

After constructing and fully initializing a configuration object it should become immutable. This is a property that has to be attached to your configuration object.

Each thread should have as easy access as possible to its thread-specific configuration data. Such data could be implemented by putting parameters into a generic collection, by sub classing the global configuration class or by providing two different configuration objects, one global and one thread-specific. The first case is not desirable due to working with generic, type-unsafe data. With the second case in place the Singleton pattern returns a common type to any caller. Each caller had to either store the retrieved configuration object in a typed class attribute, or it had to query the Singleton for the configuration each time it is needed. The developer is not hindered to proceed as the latter when working with a Singleton. Unclean design may become an issue. The third case, having two configuration objects, is not realizable with a Singleton as it could only return a single object. A wrapper object had to be provided as a workaround. The design may degenerate as well.

In case serialization of the object tree providing a thread's functionality is needed, the configuration is not serialized when it is not stored directly in one of the thread's classes. This may happen if the configuration is accessed by querying a possible Singleton each time. To get to

a serialization containing the configuration manual work would be required when using the classic solution. Finally we end up with a reference to the configuration within a threaded class:

In general, static methods or the Singleton pattern itself may not be desirable, depending on the individual developer's perspective and imagination of a clean design. Singleton, for example, is often seen as an antipattern (also compare [6] for some arguments against Singleton).

2.1 Forces

Summing up the consequences, the following forces can be identified:

- Application-specific as well as thread-specific configuration data is needed.
- Configuration data must be completely initialized before it can be accessed.
- Reading the configuration may be a time-consuming task, thus it should not be executed more than once.
- The configuration data should be read-only (immutable) after initialization.
- Access to the configuration data should be as easy as possible.
- Serialization of objects relying on configuration data should automatically include the configuration.
- The solution should enable the developer to achieve a clean design.

3 Solution

Provide an ordinary (instantiable) class that stores thread-specific configuration information. This is called the *Thread Configuration Provider* (short: *ThreadConfigProvider*). Pass the same instance of a certain thread configuration object to any other object of the same thread that needs access to this configuration.
Also provide an instantiable class holding global (application-wide) configuration information. This is called the *Application Configuration Provider* (short: *AppConfigProvider*). An instance will be passed to the *Thread Configuration Provider* at its construction.

The actors of this pattern are displayed in the class diagram in figure 1.

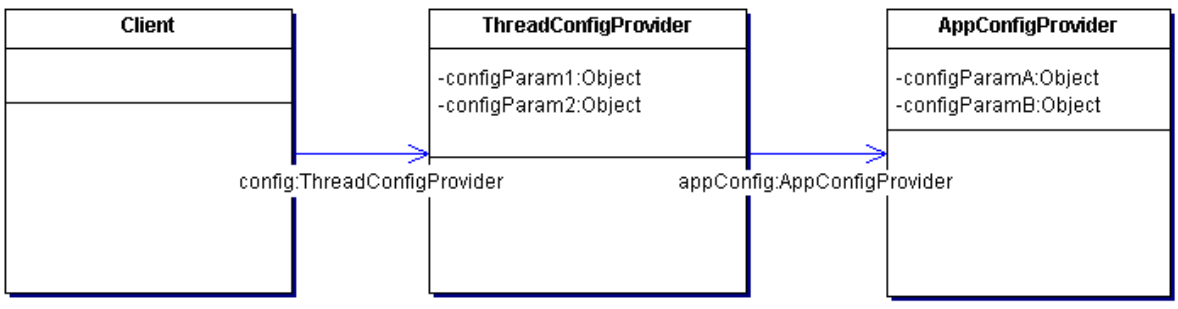

Figure 1: Class diagram of the pattern.

A *Client* class that has to have access to the configuration gets passed a reference to a *ThreadConfigProvider* at construction. The *ThreadConfigProvider* in turn keeps a reference to an *AppConfigProvider*. The overall workflow is shown in the sequence diagram in figure 2:

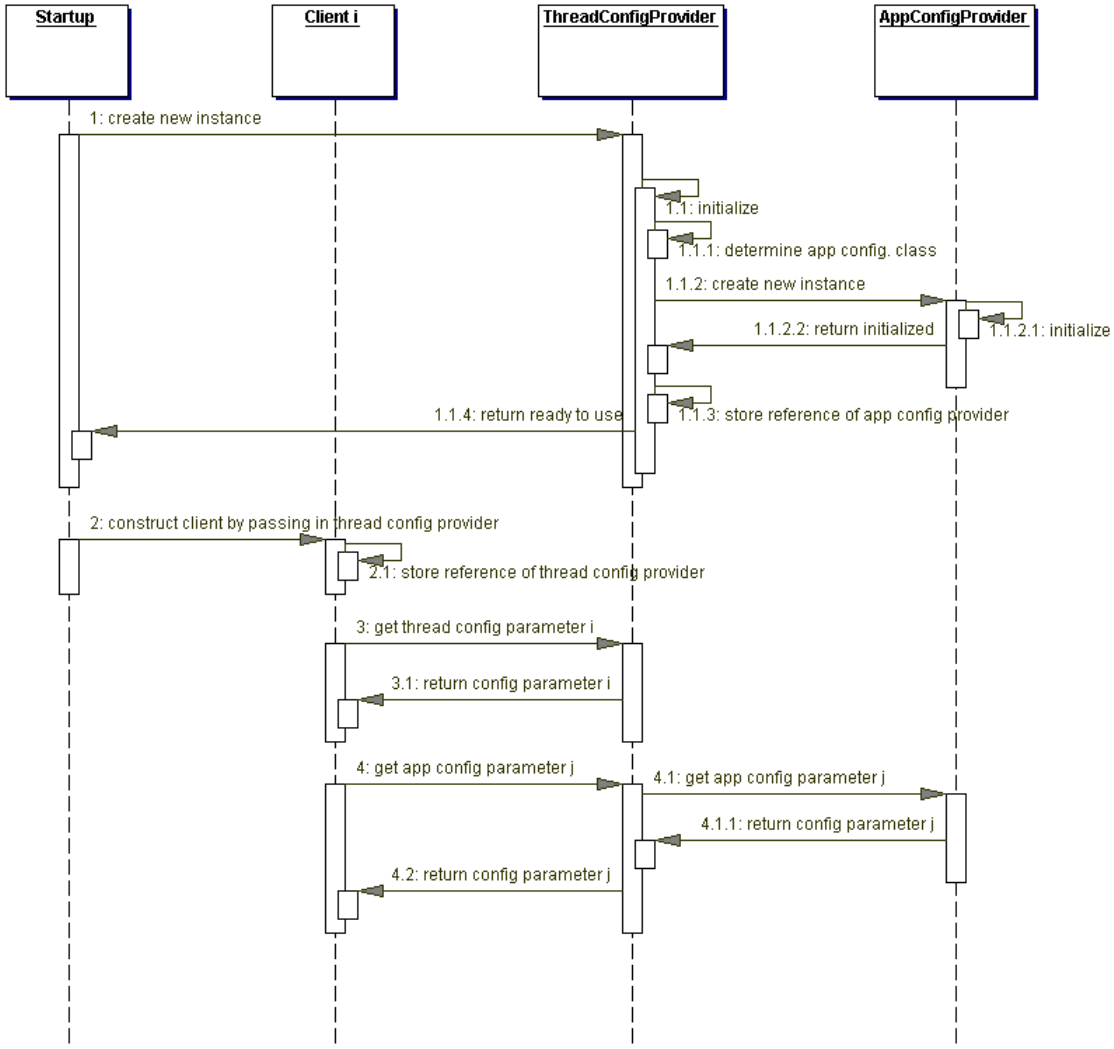

Figure 2: Sequence diagram of the pattern.

In the above figure, a *Client* class is any class needing a reference to the configuration. The above sequence diagram indicates with “Client i” that more than one *Client* class exists, which is the typical case. Each *Client* gets constructed the same way during system start-up as shown in the figure. The system’s start-up is individual to each application. It is only important that the class invoked to start-up the system cares about the construction of the configuration as early as possible to make it accessible as soon as possible to the whole application. An altnerative to passing *AppConfigProvider* to the constructor of *ThreadConfigProvider* is indicated in figure 2. Proceeding that way would make sense in case instantiation of *AppConfigProvider* should be encapsulated, e.g. by package protected access to the class.

4 Consequences

This pattern brings the developer several benefits, contrasted by some liabilities. The benefits mainly result from the straight-forward way configurations are implemented, i.e. as ordinary objects without static methods. The main origins for the benefits and liabilities are the forces the pattern puts on the class design, and the logic that is necessary to ensure the consistency of a global configuration object.

4.1 Benefits

- Enforces a complete setup at start-up of the application as there is the need for having a completely setup configuration available before any *Client* dependant on it can be constructed.
- Avoids the need for using locks for ensuring a correctly initialized configuration object before accessing it the first time.
- Reading in the configuration twice is prevented.
- Grants threads straight-forward access to their specific as well as to a global configuration which makes it easy to access configuration data.
- Avoids static methods and Singletons which may lead to a cleaner design.
- Makes serialization of classes relying on configuration more easily.
- Permits the easy identification of configurable objects by inspecting each class's constructor.
- When cloning a thread's class relying on configuration data, the implementation of the clone mechanism has to ensure that the clone is initialized with the configuration data if the class instance to duplicate stored the configuration in a local attribute. This can be accomplished quite easily by the developer but at first it is a source of errors that earliest is recognized when actually doing the cloning and furthermore executing a clone's function that relies on configuration data.

4.2 Liabilities

- For existing classes it may break their interface if an additional parameter for the *Configuration Provider* is introduced to their constructor(s). Passing in the configuration provider to the *Client* later than on construction (e.g. by using a setter) bears the danger that the configuration is not available with the *Client* when needed. This case may only come into play if a relevant part of the application's source code cannot be changed. Relevant parts normally contain *Client* classes only. Third party classes are not to be considered as they work already without configuration and will not have to be integrated into a configuration process.
- Cloning a configuration may become more difficult because no two configurations of the same type (e.g. local, global) could be permitted to exist in the same scope (thread, application). This assumption may come into play when implementing a locking mechanism for making the configuration read-only.
- Inheriting from a *Configuration Provider* forces the developer to ensure that a locking mechanism of the inheriting class is synchronized with the one of the super class. This is a source for errors and raises the effort necessary.

5 Implementation

5.1 Providing a global configuration instance

It is suggested passing a reference of a configuration object to the *ThreadConfigProvider*. Constructing a sample *Client* class could look like as shown in figure 3:

```
// System start-up: construct global configuration once.
AppConfigProvider globalConf = new AppConfigProvider();

// Setup global configuration parameters.
globalConf.setParamX("Y");

// Setup parameters of thread-specific configuration #1.
ThreadConfigProvider conf1 = new ThreadConfigProvider(globalConf);

conf1.setParam1(42);
conf1.setParam2("XYZ");

// Construct a client and provide the global configuration.
Client1 client1 = new Client1(conf1);

// Do something with client1 here.
...

// Setup parameters of thread-specific configuration #2.
ThreadConfigProvider conf2 = new ThreadConfigProvider(globalConf);

// Construct 2nd client with different thread config but same global config.
Client2 client2 = new Client2(conf2);
```

Figure 3: Setting up a configuration and dependent client classes.

It may seem curious at first glance passing the same instance of an object to all classes needing access to it. This means adding an additional parameter to the constructor of each such class. But the issue can be resolved after looking at some advantages of the solution:

- Having a direct reference to an object is the easiest way accessing it.
- It is immediately clear when the configuration is setup and accessible to an object, i.e. just when the configuration instance is passed to the object.
- The application's design is directed towards a cleaner way because the configuration has to be setup first after it can be used.
- Passing an object's instance to a constructor can be automated via the principle of *Inversion of Control* (see section 6).

5.2 Ensuring consistency of the central instance

With a single global instance passed to multiple objects, the question arises: How can it be ensured that the global instance is not changed by an object (here: thread) having access to it? Several solutions are possible to make the global configuration instance immutable:

1. Do not provide setters or global attributes for configuration parameters in the configuration object. This seems inconvenient when setting up the configuration.
2. Verify that the caller of a setter is valid, e.g. by evaluating the stack.
3. Allow locking the configuration after it has been setup and do not permit further changes to the configuration after it has been locked.

Aspect three in the above list seems the cleanest solution while the second one could be added to lay over an additional authorization. Aspect one narrows down possible scenarios too much and will be discussed shortly in the variants section.

It should be noted that even if the Singleton solution discussed at the beginning of the paper was used, these issues came into play.

An example code illustrates the solution up to now in more detail (figure 4):

```
public class StartApplication {
  public static void main(String[] args) {
    ThreadConfigProvider conf = new ThreadConfigProvider();
    conf.setUp();
    conf.lock();
    Client client = new Client(conf);
    client.beginWork();
  }
}
```

```
public class ThreadConfigProvider() {
  private int m_param1; // any configuration parameter here
  private String _param2; // ditto

  private boolean m_isLocked; // is the configuration locked?

  public void setUp() {
    // Set up the configuration by provider parameters if necessary.
    // This could also be done from outside the configuration via the
    // below setters.
  }

  public void lock() {
    m_isLocked = true;
  }

  public void setParam1(int param1) {
    if (m_isLocked) {
      throw new IllegalStateException("Configuration already locked.");
    }
    m_param1 = param1;
  }

  // Same as above for param2 (skipped here)

}
```

```
public class Client() {
  private ThreadConfigProvider m_conf;

  public Client(ThreadConfigProvider a_conf) {
    m_conf = a_conf;
  }
}
```

Figure 4: Source code for Configuration Provider (basics for three classes).

In case it may be necessary to dedicatedly change the configuration after it has been locked, an explicit and secured unlock mechanism could be provided. This temporary unlock could be feasible for single parameters as well as for the whole configuration (compare JGAP [1] for a possible implementation). Adding an optional unlock-checking guard with elective timeout to all parameter-returning getter methods of the configuration permits the getters to wait until the configuration is re-setup completely.

6 Variants

Configuration Provider is extendable in some ways. Actually this is an expected property of the pattern as no static methods are used. This paper introduced the main concept of the pattern. Functional extensions exist in quite a large number.

6.1 Facilitate construction of configuration-dependent objects

This paper proposes to pass the relevant configuration information at construction of each object relying on it. To facilitate this construction, a so-called *IoC* container (IoC = *Inversion of Control*, also called *Dependency Injection*) could be used. The *IoC* container is capable of building a construction tree of all classes involved when constructing a specific class. The involved classes are determined by the references of the class to be constructed itself as well as by considering its children and the children of the children.

6.2 Additional configurations

Besides a thread and an application configuration, other configurations could exist as well. The implementation is straight forward as only new and independent classes have to be introduced which could be referenced by the *Thread Configuration Provider* or passed to it at construction time.

6.3 Configuration objects without setters

As shortly described in section 5.2, a way of making an object immutable is skipping public attributes and setters (the mechanisms of introspection and reflection to still overcome immutability are not discussed here). Then, to setup the configuration, a single mechanism is delivered: the constructor-based initialization. Among other possibilities, a good way of providing the configuration data is by passing in a name of a file holding the data. The

configuration object can then read in and parse the file and store the information in its private attributes.

6.4 Direct access to configuration object

Skip passing a reference of the configuration object to thread classes and access it by directly reading out system properties where needed. This may work when only a few classes have to access the configuration. But it becomes annoying reading out a system property in the same way for many classes.

7 Known Uses

Known uses of the pattern or a similar form include

- JGAP [1] implements a *Configuration Provider* by passing a convenience object to all objects needing access to the configuration. The convenience object returns an instance (that is unique within a thread) of the *Configuration Provider*.
- The Excalibur framework [3] from Apache provides an interface named *Configurable*. Any class implementing it realizes a method called *configure*. Via this method – not at construction – an instance of a configuration object is passed to the configurable object.
- FreeMarker [4], a template engine, allows to set a configuration object for each template. It is up to the developer to either let multiple templates share a common configuration instance or to grant each template a unique configuration.

8 Related Patterns

Configuration Provider has some aspects in common with other patterns. These patterns will be shortly described in this section to differentiate them from *Configuration Provider*.
The well-known Singleton ensures that only one instance of an object exists. The typical Singleton implementation is often regarded as a design flaw as it relies on a static method *getInstance()* which constrains inheritance and often is implemented in a way which has problems with concurrent access.
A *Data Transfer Object* (short: *DTO*) [5] is commonly used as a data container for transferring data in a distributed environment. In contrast to *Configuration Provider* a *DTO* can be passed to an object at virtually any time. It is not meant to be accessible by multiple objects at the same time. *DTO* itself may be seen as part of *Configuration Provider* in that a *DTO* serves as a (dumb) data container that could hold configuration data. *DTO* does not contain mechanisms to ensure consistency or global accessibility.
Thread-Specific Storage [5] provides each thread with an object that only the specific thread can write to. Other threads can concurrently read out the object as well. A typical use case is keeping a variable with the number of an error occurred during processing a specific thread.

References

[1] Meffert, K. et al.: JGAP: Java Genetic Algorithms Package. http://jgap.sf.net, 2007.

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

[3] Apache Excalibur: http://excalibur.apache.org/framework/index.html.

[4] FreeMarker: http://freemarker.sourceforge.net/.

[5] Buschmann, F; Henney, K; Schmidt, D. C.: Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4. Wiley, 2007.

[6] Henney, K.: One or Many. Javaspektrum, 2003.