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-ilmena.de*[2]

Abstract

You have the requirement to configure different aspects of your multi-threaded application. This includes configuring threads specifically as well as providing general application parameters. The implementation should not be based on static methods in order to keep it extendable, reusable and to allow multiple threads to operate with it . This paper introduces the pattern *Configuration Provider*. It offers a central instance for configuration aspects that resolves the mentioned issues and results in easily extendable classes without static methods.

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

1 Context

1.1 Thumbnail

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

1.2 Use Case

Multi-threaded applications that rely on global as well as on thread-specific configurations.

2 Example

An application can use threads to perform specific tasks. Each task should be configurable separately by the application. Besides, each thread should have access to a single global configuration.
Part of a thread-specific configuration could be a work item a thread should care about exclusively. All parts of each thread should have access to the application's configuration. Each thread itself must furthermore access its specific configuration, namely the work item's configuration particular to the thread. An example for part of a global configuration is a directory in which to put log files. Such a directory should be unique among an application.
One way of providing global access to configuration data is using static methods to have global access to the configuration. In a threaded application things are more complicated. If each thread additionally requires its own configuration, the configuration must be stored and retrieved via the configuration object by considering the ID of each calling thread.

granted to EuroPLoP for use with conference

In summary, 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.
Using static methods is not feasible because of problems arising from concurrent access. Besides, static methods prevent inheritance and produce difficult to maintain code.

3 Problem

Multiple objects in an application have to be configured consistently. An application running multiple threads provides each thread with a thread-specific configuration as well as with the global application configuration. The application has to provide the threads with the configuration because thread-specific configurations are set-up outside the threads and the global configuration has to be consistent. All objects within a thread relying on the thread or application configuration should be able to easily access it.
How do you implement such a solution? How can you make available nonstatic methods of an object application-wide?

The solution is influenced by the following forces:

- All objects within a thread should have access to a unique thread configuration whereas they should also be able to consistently access a global application configuration.
- A consistent configuration is determined at application-level, e.g. during start-up or via a user dialog. This forces the application to pass in the configuration to each thread instead of having each thread determining its own configuration.
- The solution should allow a clean implementation in Java; hence static methods should be avoided to access configuration objects.
- It must be ensured that no two threads access the same configuration object at the same time.
- Extending a static method is not possible (at least with Java). Thus static methods cannot be used in case inheriting a configuration class is necessary.
- Extending the configuration object should be possible without touching other classes.
- Extending the configuration should leave to a consistent state so that global configuration data is available to all accessors as the same data and not as different configurations.
- The configuration object should care that its state cannot become invalid.

4 Solution

Provide an instantiable and immutable class that stores thread-specific configuration information. This is the *Thread Configuration Provider*. Pass the same instance of the thread configuration object to any other object of the current thread that needs access to the thread configuration. All objects within the same thread share the same *Thread Configuration Provider* which is unique among the application.
Also provide an instantiable class holding global (application-wide) configuration information.
This is the *Application Configuration Provider*. An instance will be passed to the *Thread Configuration Provider* at its construction.

4.1 Structure

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

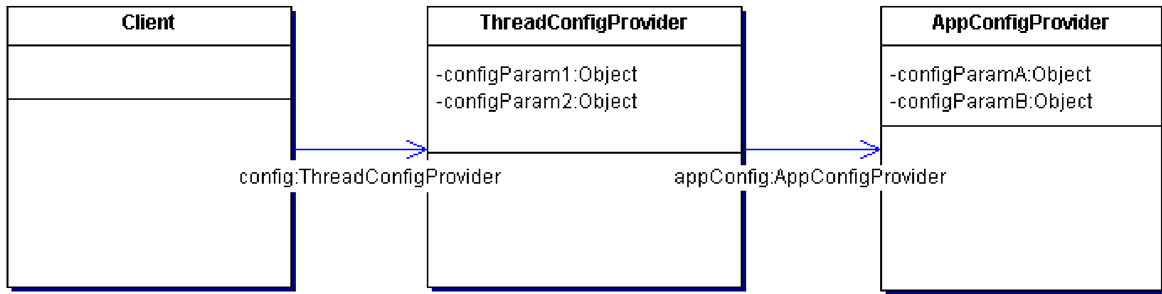

Figure 1: Class diagram of the pattern

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:

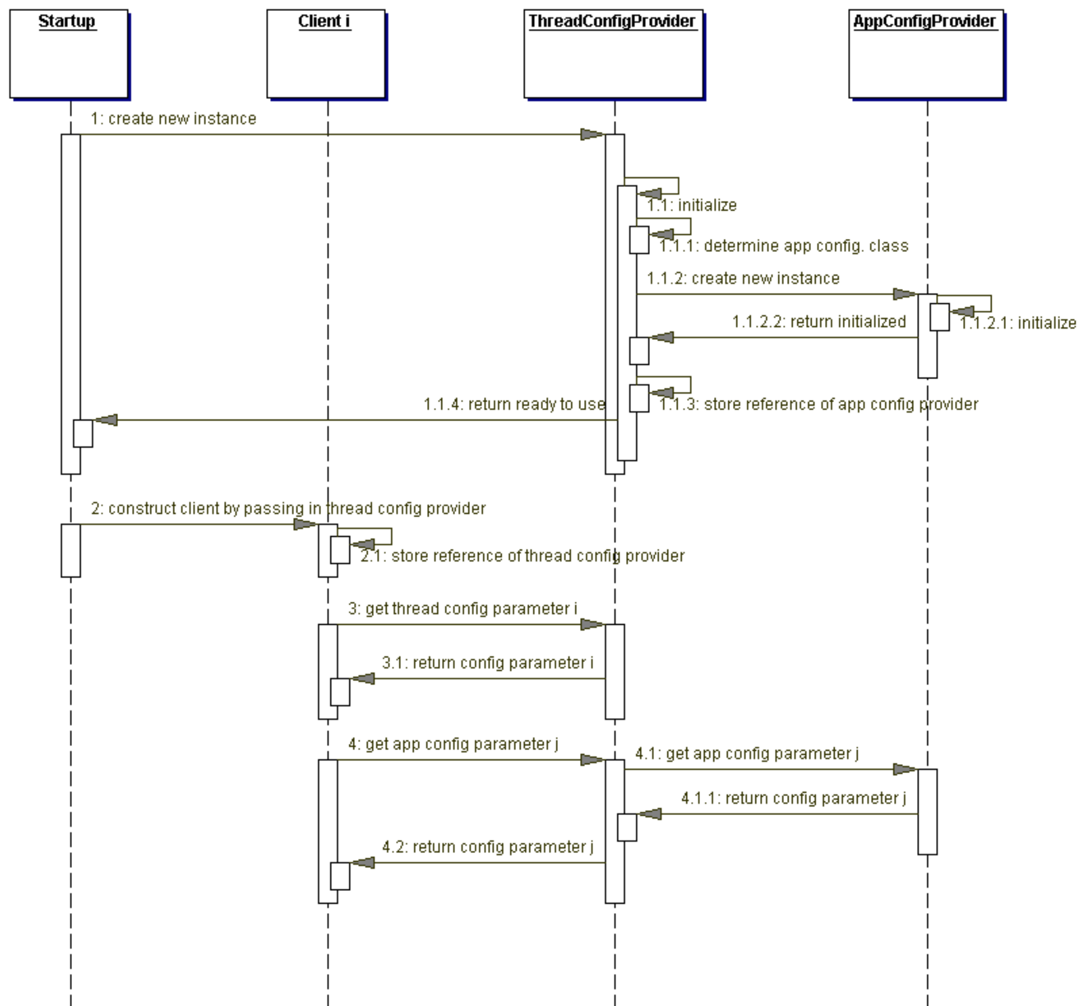

Figure 2: Sequence diagram of the pattern

The above sequence diagram indicates with “Client i” that more than one *Client* class is existent, which is the typical case. A *Client* class is any class needing a reference to the configuration. 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.

4.2 Implementation

4.2.1 Providing a central configuration instance

It is not possible to use static methods to implement a central instance for application-wide access because different threads need access to different central instances. Thus, passing a reference of a configuration object is suggested. In figure 2, it is identified as *ThreadConfigProvider*. A sample *Client* class would be constructed as shown in figure 3:

```
// System startup: construct configuration.
ThreadConfigProvider conf = new ThreadConfigProvider();

// Now setup configuration.
conf.setParam1(42);
conf.setParam2("XYZ");

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

// Do something with client1 here.
...

// Construct another client and provide the same configuration.
Client2 client2 = new Client2(conf);
```

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 the advantages of the solution:

- Each class that needs access to the configuration can be identified easily by looking at its constructor.
- Having a 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 7.4).

4.2.2 Ensuring consistency of the central instance

With a single central instance passed to multiple objects, the question arises: How can it be ensured that the central instance is not changed by an object having access to it? Several solutions are possible to make the central 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. 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 the configuration 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."
                                     + "It is not allowed changing it.");
    }
    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.

4.2.3 Introducing application configuration

As shown in figure 2, application-wide configuration data is suggested to be stored in the *ThreadConfigProvider*. A solution that builds on providing the class name of the application configuration via a system property could look like this:

```
public class StartApplication {
  ...
  // Define class for providing application configuration
  System.setProperty(ThreadConfigProvider.APPCONFIGCLASSNAME,
                     "my.package.AppConfig");

  // Setup the configuration
  ThreadConfigProvider conf = new ThreadConfigProvider();

  // Construct Clients
  ...
}
```

```
public class ThreadConfigProvider {
  public final static String APPCONFIGCLASSNAME = "appConfigClassName";

  private IAppConfigProvider m_appConfig;

  public ThreadConfigProvider() throws Exception {
    String className = System.getProperty(APPCONFIGCLASSNAME);
    m_appConfig = (IAppConfigProvider)Class.forName(className).newInstance();
  }

  // Rest of class as shown in section 4.2.2
}
```

Figure 5: Source code for Configuration Provider (extension)

With no limitations it would be possible to set the system property from the command-line instead of doing this in a class like *StartApplication*:

```
java –DappConfigClassName="my.package.AppConfig" StartApplication
```

Figure 6: Setting a configuration provider from the command line

5 Example resolved

The example from section 2 can be resolved by applying *Configuration Provider*. The proposed solution looks like this:
Each object that must have access to the configuration gets it passed in at construction. This means that the configuration has to be provided readily as early as possible at system start-up. Because the configuration is locked after is has been setup, no *Client* can change the configuration then, thus no inconsistencies can arise. Having a reference to the configuration at hand in each *Client* class means straight-forward access to the configuration with the easy ability of identifying each *Client* class that depends on the configuration.
The global application configuration and the local thread configuration are stored in separate classes. The latter one references the former one.
Because no static methods are necessary to implement a globally accessible configuration, the configuration classes can be extended easily by subclassing them.

6 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.

6.1 Benefits

- 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 leads to a cleaner design.
- Decouples the application's configuration from business classes which makes extensions easier.
- Distincts between task-specific (local) and application-wide (global) configuration data. Enables a cleaner design.
- 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.
- Makes reuse of the implemented configuration classes in different contexts and applications possible as extracting abstract base classes and introducing additional sub classes does not get in conflict with typing of the configuration object.
- Permits the easy identification of configurable objects by inspecting each class's constructor.

6.2 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 be difficult because no two configurations of the same type (e.g. local, global) should exist in the same scope (thread, application).

- Inheriting from a *Configuration Provider* forces the developer to ensure that the locking mechanism of the sub class is synchronized with the one of the super class. This is a source for errors and raises the effort necessary.

7 Variants

Configuration Provider is extendable in many 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.

7.1 Direct access to configuration object

Skip passing a reference to the configuration object and access it via 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.

7.2 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 will be referenced by the thread configuration.

7.3 Configuration objects without setters

As shortly described in section 4.2.2, 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). 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.

7.4 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) 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.

8 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.

9 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 has problems with concurrent access.

A Data Transfer Object (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 (dump) 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] JGAP: Java Genetic Algorithms Package. http://jgap.sf.net.

[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.