

The New Moses Repository Architecture

Kim Mason

Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
kim@tik.ee.ethz.ch

September 21, 2000

1 Why Create a New Repository Architecture?

The new Moses repository architecture was created to fill the need for a simple repository that could be inspected and repaired by the user in case of failure, and easily maintained because of its simple design. The new repository provides minimal storage and directory services, with the expectation that other services will be placed at a higher level.

1.1 What Does the New Repository Look Like?

The new repository is largely based on a file system. The repository has a root directory, and sub-directories can be created, existing objects created, moved, or re-named. A repository is represented by a class implementing the Java interface `moses.rep.Repository`.

The `moses.rep.Repository` interface provides some additional services, including providing an associated set of types (Strings) for each contained object, and providing a *Repository Type-Serializer-Map* that maps type names to instances of `moses.rep.Serializer` and `moses.rep.DeSerializer`.

2 Types and Object Access

The repository allows us to store Java objects in a persistent manner. This requires some form of serialization. The Java serialization mechanism is flawed, as it may fail if classes are modified, and serialized objects are not in a useful form. For this reason, we use a custom serialization mechanism.

Every serialized object in the Repository is associated with a set of types – some standard types are defined in `moses.rep.RepositoryConstants`, and are MIME types (Strings). Custom types can be added easily. A set of types is required because a serialized object may be interpreted in several ways. For example, a petri-net may be an instance of a `moses.graph.Graph`. This may be serialized as an XML document. For this reason, the serialized object would have the types `text/plain`, `text/xml`, `application/x-moses-graph`, `application/x-moses-petrimet` (note that the `graph` and `petri-net` aren't defined, but have been created for this example).

2.1 Types and Java Classes

There should be a fixed relationship between the Java classes that a `Serializer` or `DeSerializer` can consume, and the type associated with that `Serializer` or `DeSerializer`. In general, a `Serializer` for a type can only consume a small set of classes or a single class, and a `DeSerializer` can only produce a single class.

2.2 Serializers and Deserializers

We require a way to get from an instance of some Java object, to a serialized object and a set of associated types. We shall continue to use the above Petri-net example. We know that we wish to serialize the object to an `application/x-moses-graph`. We would ask the *Type-Serializer-Map* for a set of `Serializer` classes that can serialize this type. We can then choose which `Serializer` to use, based on the result of its `getTypes()` method (this will tell us that it is also serialized to XML and text), or perhaps the class name. We would then call the `Repository.put` method, providing the object to put, its location, an instance of the `Serializer`, and the Set of extra types (in this case, a 1 element set containing `application/x-moses-petrinet`).

2.3 Deserializing

De-serialization requires us to ask the *Repository Type-Serializer-Map* for a set of `DeSerializers` that will de-serialize to the type that we desire. We then provide the input-stream to the `DeSerializers.canDeserialize` methods, and finally to the `deserialize` method when the appropriate `DeSerializer` is found.

2.4 The Type-Serializer-Map

The *Type-Serializer-Map* for the repository can be accessed through the `Repository` interface. The `FileSystemRepository` additionally allows a default type map to be provided in its constructor. Within the Moses framework, this default type map is derived from a file named `DefaultTypeMap.properties`.

2.5 Provided Repositories

A `Repository` that is implemented on top of a normal file-system is provided. This is the `moses.rep.FileSystemRepository`.

3 Tools

A tool architecture is layered upon the `Repository` architecture. Tools are created by a `moses.tool.ToolFactory`. A representation of a tool factory can be created within the repository, providing information such as the Java class that implements the tool factory, initialization parameters for that class, and commentary information. `Serializers` and `DeSerializers` are provided in `moses.rep.serializers` that allow the reading and writing of a `ToolFactory` repository representation.

A `ToolFactory` allows instances of tools to be created, and instantiation parameters to be provided. The `ToolFactory` is first instantiated using the `initialize` method, after which the `canCreate` and `create` methods may be called – these

methods are provided with parameters that are used to instantiate the tool itself, including a graphical context if necessary.

It is intended that objects such as `moses.rep.RepositoryItems` may be passed to the tools as their initialisation parameters. The tools may display on-screen using the graphical context passed to them.

4 Provided Repository User Interface

A user interface is provided that allows access to `FileSystemRepositories` is provided. There are 3 parts to this user interface - the *Repository Tree*, the *Tool Pool* and the *Parameter List*. Initially, a repository may be created using the `File->New` menu entry. Projects may be created within the repository using the right mouse button, and elements likewise removed.

The *Tool Pool* and *Parameter List* are strongly associated. Elements may be selected within the repository (ctrl and shift allow the selection of multiple elements), and copied using a right mouse click on the selected elements (a popup menu will appear, in which copy can be selected). Representations of these repository elements can then be pasted into the *Parameter List* using the paste button. It is expected that these are the ordered parameters that a tool will consume when it is launched.

The *Tool Pool* provides a list of tools available in the repository. It has a search path which is used to search for tools, and a graphical representation is displayed. The tools can be selected, and launched - the parameters in the *Parameter List* will be provided to the tool, and it shall be launched.

5 Notes to Rob

This is still a little incomplete. For example, I'm currently implementing a 'tool creator', which will be accessible from the main repository tree, allowing the user to easily create tool descriptions. The repository tree is also in need of a 'refresh' item. I can also make it such that if the provided parameters don't suit a tool, the tool will be grayed out (the `ToolFactory` has a `canLaunch` method). I'm doing these now, but I figured that I'd send this document immediately.

The idea is that once the tool creator is in place, tools will be used to do everything else, including create other objects. For example, if you wish to create edit a component within a directory, you would start the editor tool with that directory provided as a parameter. This is to avoid creating very specific sets of interfaces within Moses. Having said this, it may be better to have a 'New Object' item on projects within the repository browser, and have some kind of object creator interface, but this doesn't seem strictly necessary at the moment.

If you would like to create your own tool descriptions, the format can be seen in `moses.rep.serializers.ToolFactorySerializer`, and would be something like

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Tool Representation 1.0 -->
<toolrep version="1.0">
  <comment>A Sample Tool Description</comment>
  <executableclass>moses.tools.TestTool</executableclass>
  <list>
    <paramname>Param1</paramname>
```

```
    <paramvalue>"Value1"</paramvalue>  
  </list>  
</toolrep>
```

I will also send you a sample repository that has a tool description within it.