SOL Software d.o.o.
www.sol.rs

Public

SOLoist Automation of Class IDs Assignment

Project: SOLoist V4
Document Type: Project Documentation (PD)
Document Version: 1.0
Date: 01.06.2015

Introduction

This document explains how class and object IDs are formed, how they were managed, and how they are supposed to be managed by SOLoist developers in the future. In short, in previous versions of SOLoist class IDs were manually specified by SOLoist developers. From the version 1.0.8, this process is automated. Now, SOLoist developers are freed from having to think about this issue, except from in extremely rare occasions when class IDs can still be manually adjusted.

Format of Class IDs

This section describes how SOLoist assigns object ID ranges to classes. These ranges are stored in a file usually named `classIDs.conf`.

Object ID is a 64-bit value and it consists of the class code and object code. Each class in the model must have its own object ID range. There must not exist two classes that have overlapping ranges. A SOLoist exception is thrown during SOLoist initialization if this situation occurs.

For example, let us take a look at the object ID of object x which is an instance of class X. If class X code occupies 10 bits in this object ID, then object code occupies 64 – 10 = 54 bits. Implicitly, there can be no more than 2^{54} = 18,014,398,509,481,984 instances of class X persisted in the database.

Class A1 code is (0x000). The first object within class A1 range:															
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
The last object within class A1 range:															
0000	0000	0011	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111
Row inside classIDs.conf:															
`000/10 = rs::sol::TestProject::A1`															

Class A2 code is (0x004). The first object within class A2 range:															
0000	0000	0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
The last object within class A2 range:															
0000	0000	0111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111
Row inside classIDs.conf:															
`004/10 = rs::sol::TestProject::A2`															

Class AN code is (0xFFC). The first object within class AN range:															
1111	1111	1100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
The last object within class AN range:															
1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111
`FFC/10 = rs::sol::TestProject::AN`															

Class codes and object codes can have different lengths. It is a tradeoff between the lengths of class and object codes. If a class code occupies less bits, it allows for more instances of that class. However, it then reduces the maximum number of classes. On the other hand, if a class

code occupies more bits, there can be less instances of that class, but more classes can exist in the model.

In most cases the default SOLoist class IDs configuration is satisfying. That is, SOLoist will generate class codes during initialization procedure and SOLoist developer would rarely need to customize anything. In cases where extremely large number of instances of a certain class is required, or in cases where extremely large number of classes is required, SOLoist developer should customize the classIDs.conf file in order to allow that.

Automation of Class IDs Generation Process

In SOLoist version 1.0.8 automation is introduced in the process of handling class IDs, i.e. in the process of managing the classIDs.conf file. SOLoist developers are spared from any responsibility regarding class codes assignment. This process is automated and attached to the Maven build phase as a java execution goal.

The java program for class codes generation must be executed in order to either initially generate or to extend the `classIDs.conf` file. The main method of the class `AutoAttachClassIDsMain` needs to be bound to the Maven build phase higher than `compile` (in the example below the test build phase is used). In other words, this code should be placed in the `pom.xml` file:

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>exec-maven-plugin</artifactId>
	<version>1.1.1</version>
	<executions>
		<execution>
			<phase>test</phase>
			<goals>
				<goal>java</goal>
			</goals>
			<configuration>
			<mainClass>rs.sol.soloist.server.server.AutoAttachClassIDsMain</mainClass>
			</configuration>
		</execution>
	</executions>
</plugin>
```

To run the automatic class codes assignment program, execute the Maven test build phase or a later phase. This can be done from the command line or from the IDE. To run it from the command line, type the following (in the folder where pom.xml resides):

mvn test -Dmaven.test.skip=true

Refresh the current project.

To run it from Eclipse, create a new Maven Run Configuration. Right-click on the project and then: Run As -> Run Configurations… -> m2 New_configuration. Set the name of the configuration in the *Name* textbox. Then set *Base directory* by browsing the workspace (*Browse Workspace…* button) and choose the current project. After that, specify the build phase (for example *test*) in *Goals* textbox and check the *Skip Tests* checkbox. Finally hit *Apply* and *Run*. Refresh the current project. This way you have created (and executed) a run configuration for execution of automatic class codes assignment. After this, you are able to run this configuration many times by choosing *Run As* and then selecting a new configuration name. You should run it every time after you change the model by adding a new class.

It is important to note that automatic class codes generator will not remove any of the already existing records from the class IDs file. This way, SOLoist programmer is allowed to manually customize this file in cases when this is needed. The automatic generator will determine the highest class IDs range and add additional (new) ranges after it.

There are 3 optional arguments that can be provided (they are always interpreted in this order). In most cases developers will not have the need for different arguments than default ones.

- **block length** – the number of bits for the class code; the default value is 10, which means that there can be 2^{10} classes and $2^{64-10}=2^{54}$ instances of each class.
- **input file** – existing class IDs file name (the file must be on the classpath); the default value is `classIDs.conf`.
- **output file** – output file path: default is `src/main/resources/classIDs.conf`.

Example 1:

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>exec-maven-plugin</artifactId>
	<version>1.1.1</version>
	<executions>
		<execution>
			<phase>test</phase>
			<goals>
				<goal>java</goal>
			</goals>
			<configuration>
			<mainClass>rs.sol.soloist.server.server.AutoAttachClassIDsMain</mainClass>
				<arguments>
					<argument>12</argument>
				</arguments>
			</configuration>
		</execution>
	</executions>
</plugin>
```

Example 2 (Initial class codes taken from `myNewClassIDs.conf`.):

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>exec-maven-plugin</artifactId>
	<version>1.1.1</version>
	<executions>
		<execution>
			<phase>test</phase>
			<goals>
				<goal>java</goal>
			</goals>
			<configuration>
			<mainClass>rs.sol.soloist.server.server.AutoAttachClassIDsMain</mainClass>
				<arguments>
					<argument>12</argument>
					<argument>myNewClassIDs.conf</argument>
				</arguments>
			</configuration>
		</execution>
	</executions>
</plugin>
```

Example 3:

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
```

```
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.1.1</version>
        <executions>
                <execution>
                        <phase>test</phase>
                        <goals>
                                <goal>java</goal>
                        </goals>
                        <configuration>
                        <mainClass>rs.sol.soloist.server.server.AutoAttachClassIDsMain</mainClass>
                                <arguments>
                                        <argument>9</argument>
                                        <argument>classIDs.conf</argument>
                                        <argument>src/main/resources/myNewClassIDs.conf</argument>
                                </arguments>
                        </configuration>
                </execution>
        </executions>
</plugin>
```

Conclusion

In most cases the classIDs.conf file should not be managed manually by SOLoist developers. The only thing they should do is to put a section of maven code into pom.xml. In rare cases where there is a need for customization of class IDs, SOLoist developers should edit the classIDs.conf file in accordance with their needs.