databene benerator 0.8.1 manual

Volker Bergmann

Publication date: 2013-05-15

Volker Bergmann

databene benerator manual 0.8.1

Copyright © 2006-2013 Volker Bergmann

This material may be downloaded and used for own, personal education.

Distribution of substantively modified versions of this document is prohibited without the explicit written permission of the copyright holder.

Distribution of the work or derivative of the work in any form for commercial purposes is prohibited unless prior written permission is obtained from the copyright holder.

databene and benerator are registered trademarks of Volker Bergmann.

All other trademarks referenced herein are the property of their respective owners.

Volker Bergmann

Heilig-Geist-Str. 7

83022 Rosenheim

Germany

volker@databene.org

2013-05-15

Table of Contents

1 Introduction to benerator.....12
1.1 Goals.....12
1.2 Features.....12
1.2.1 Data Synthesization.....12
1.2.2 Production Data Anonymization.....13
1.3 State of the benerator.....13
1.4 Building Blocks.....13
1.5 Database Support.....14
2 Installation.....15
2.1 Download the distribution binary.....15
2.2 Unzip Benerator.....15
2.3 Set BENERATOR_HOME.....15
2.4 Optional: Install JDBC drivers.....15
2.5 Optional: Set up log4j.....15
2.6 On Unix/Linux/Mac systems: Set permissions.....16
2.7 Mac OS X configuration.....16
2.8 Verifying the settings.....16
3 The Benerator Project Wizard.....17
3.1 Starting the wizard.....17
3.2 Configuring the project.....18
3.3 Creating and running the project.....18
4 Quick tour through the descriptor file format.....19
4.1 <setup>.....19
4.2 benerator properties.....19
4.3 <include>.....20
4.3.1 Inclusion of properties files.....20
4.3.2 Sub-Invocation of descriptor files.....20
4.4 Global settings.....20
4.5 <import>.....20
4.6 <generate>.....21
4.6.1 "constant".....21
4.6.2 "values".....22
4.6.3 "pattern": Generation by Regular Expression.....22
4.7 <iterate>.....22
4.8 "offset".....22
4.9 <echo>.....22
4.10 <beep/>.....23
4.11 <comment>.....23
4.12 <execute type="shell">.....23
4.13 <wait>.....23
4.14 <error>.....23
4.15 <if>.....23
4.16 <while>.....24
4.17 <id> - Generating unique identifiers.....24

4.18 Naming Conventions........25
5 Data Generation Concepts........25
5.1 Naming........25
5.2 Entity Data........25
5.3 Simple Data Types........26
5.4 Data Characteristics........27
5.4.1 Distribution Concept........27
5.5 Generation Stages........28
5.5.1 System Initialization Stage........29
5.5.2 Precondition Checking Stage........29
5.5.3 Core Data Generation Stage........30
5.5.4 Mass Data Generation Stage........30
5.5.5 Data Postprocessing Stage........30
5.5.6 Result Validation Stage........30
5.6 Metadata Concepts........31
5.6.1 Case Sensitivity........31
5.6.2 Namespaces........31
5.6.3 <setting> and benerator identifiers........31
5.7 Benerator Components........32
5.8 Instantiating Global Components........32
5.9 Instantiating Local Components........33
5.9.1 Referral........33
5.9.2 Default Construction........33
5.9.3 Parameterized Construction........33
5.9.4 Property-based Construction........33
5.10 Descriptive Data Generation........33
5.11 Default Data Generation........33
5.12 Constructive Data Generation........34
5.13 Validating Data Generation........35
5.14 Prototype-based Data Generation........36
5.15 Sample-based Data Generation........36
5.16 Variables........37
5.17 Combining components and variables........37
5.18 Referring Files........37
5.19 Protocols........38
5.20 Relative URIs........38
5.21 Importing Entities........38
5.22 Custom Importers........38
5.23 Consumers........38
5.23.1 Specifying Consumers........38
5.23.2 Consumer Life Cycle........39
5.24 Exporting Data to Files........39
5.25 Post Processing Imported or Variable Data........40
5.25.1 overwriting post processing........40
5.25.2 "script" post processing........40
5.25.3 "map" post processing........40
5.25.4 "converter" post processing........40
5.26 Anonymizing Production Data........42
5.27 "condition"........42
5.28 Converters........43

5.29 Validators........ 44
5.30 Creating random Entities........ 44
5.31 Entity Count........ 44
5.32 Using Predefined Entities........ 44
5.33 Iterating Predefined Entities Consecutively........ 45
5.34 Applying a Weight Function........ 45
5.35 Applying a Sequence........ 45
5.36 Importing Weights........ 45
5.36.1 Importing primitive data weights........ 45
5.36.2 Weighing imported entities by attribute........ 46
5.37 Nesting Entities........ 46
5.38 Imposing one-field business constraints........ 47
5.39 Imposing multi-field-constraints........ 47
5.40 Default Attribute Settings........ 47
5.41 Settings........ 48
5.42 Querying Information from a System........ 48
5.43 Attribute Metadata Reference........ 49
5.43.1 Descriptive Attribute Metadata........ 49
5.43.2 Constructive Attribute Metadata........ 49
5.44 Scripts........ 50
5.44.1 this........ 51
5.45 Handling Errors........ 51
5.45.1 onError........ 51
5.45.2 BadDataConsumer........ 51
6 Regular Expression Support........ 52
6.1 Characters........ 52
6.2 Character Classes........ 53
6.3 Operators........ 53
6.4 Frequently asked Questions........ 53
7 Processing and creating CSV Files........ 55
7.1 Iterating entity data from a CSV file........ 55
7.2 Creating CSV files........ 55
8 Using Relational Databases........ 56
8.1 Import and <database>........ 56
8.2 Usual Database Settings........ 57
8.3 Using Database Repositories........ 57
8.4 Caching Database Metadata........ 58
8.5 Executing SQL statements........ 58
8.5.1 Alternative Delimiters........ 58
8.6 Inserting entities into a database........ 59
8.7 Database-related Id Generators........ 59
8.7.1 SequenceTableGenerator........ 59
8.8 Handling of common Columns........ 60
8.9 Determining attribute values by a database query........ 61
8.9.1 Cycling through query result sets........ 61
8.9.2 Applying a distribution to a query........ 61
8.9.3 Sub selectors........ 61

8.10 Resolving Database Relations.......... 62
8.10.1 Automatic referencing.......... 62
8.10.2 Null references.......... 62
8.10.3 Selective Referencing.......... 62
8.10.4 Other referencing options.......... 62
8.11 Composite Keys.......... 63
8.12 Prototype Queries.......... 63
8.12.1 Prototype Queries on Entities.......... 63
8.13 Exporting Database Content.......... 63
8.14 Updating Database Entries.......... 64
8.15 Controlling Transactions.......... 64
8.16 Transcoding Database Data.......... 65
8.16.1 Copying database entries.......... 65
8.16.2 Restricting the copied Set.......... 65
8.16.3 Transcoding.......... 66
8.16.4 Cascaded Transcoding.......... 66
8.16.5 Merging Foreign Key Relations.......... 66
8.16.6 Defining Identities.......... 67
8.16.7 Limitations.......... 68
9 Using mongoDB.......... 69
9.1 mongo4ben Installation.......... 69
9.2 Usage.......... 69
9.3 Examples.......... 69
10 Generating XML Files.......... 71
10.1 Schema-based XML file generation.......... 71
10.1.1 Introduction.......... 71
10.1.2 Configuring Attribute Generation.......... 74
10.1.3 Using <variables> in XML Schema.......... 74
10.1.4 Importing Properties File Data.......... 76
10.2 Generating XML in classic descriptor files.......... 76
10.2.1 Using data types from XML schema files.......... 76
10.3 Conclusion.......... 76
11 Advanced Topics.......... 77
11.1 JavaBeans and the Benerator Context.......... 77
11.2 Importing Java classes.......... 78
11.3 Looking up services/objects via JNDI.......... 78
11.4 Calculating global sums.......... 78
11.5 Querying information from a system.......... 79
11.6 The MemStore.......... 79
11.7 Datasets.......... 80
11.7.1 Region nesting.......... 82
11.8 Chaining generators.......... 82
11.9 Invoking Benerator programmatically.......... 82
11.9.1 Invoking Benerator as a whole.......... 83
11.9.2 Making Benerator generate data and invoke a custom class with it.......... 83
11.9.3 Using generators defined in a descriptor file.......... 83
11.10 Tasks.......... 84
11.11 Staging.......... 84
11.12 Template data structures.......... 85
11.13 Generating arrays.......... 86

11.14 Scoped Generation.... 86
11.14.1 Default (implicit) scope.... 86
11.14.2 Explicit scope.... 87
11.14.3 Global scope.... 88
11.15 Composite Data Generation.... 88
11.16 Composite Data Iteration.... 89
12 Generating Unique Data.... 90
12.1 ID Generation.... 90
12.2 Unique Number Generation.... 90
12.3 Unique String Generation.... 90
12.3.1 Uniqueness with Regular Expression.... 90
12.3.2 Making Strings unique.... 90
12.4 Removing Duplicate Values.... 90
12.5 Unique iteration through a source.... 91
12.6 Unique Generation of Composite Keys.... 91
12.6.1 Simplistic Approach.... 91
12.6.2 Cartesian Product.... 91
12.6.3 Prototype Approach.... 92
12.7 Achieving local uniqueness.... 92
13 Scripting.... 93
13.1 Shell scripting.... 94
14 DatabeneScript.... 95
14.1 Motivation.... 95
14.2 Examples.... 95
14.2.1 Variable definition.... 95
14.2.2 Variable assignment.... 95
14.2.3 Object construction.... 95
14.2.4 Date arithmetic.... 95
14.2.5 Java integration.... 95
14.3 Syntax.... 95
14.3.1 Comments.... 95
14.3.2 White Space.... 96
14.3.3 Data Types.... 96
14.3.4 Identifiers.... 96
14.3.5 Escape Sequences.... 96
14.3.6 String Literal.... 96
14.3.7 Decimal Literal.... 96
14.3.8 Integral Number Literal.... 96
14.3.9 Boolean Literal.... 96
14.3.10 null Literal.... 97
14.3.11 Qualified name.... 97
14.3.12 Constructor invocation.... 97
14.3.13 Bean construction.... 97
14.3.14 Method invocation.... 97
14.3.15 Attribute access.... 97
14.3.16 Casts.... 97
14.3.17 Unary Operators.... 97
14.3.18 Arithmetic Operators.... 97
14.3.19 Shift Operators.... 98
14.3.20 Relation Operators.... 98
14.3.21 Equality Operators.... 98
14.3.22 Bitwise Operators.... 98
14.3.23 Boolean Operators.... 98
14.3.24 Conditional Expression.... 98

14.3.25 Assignment... 98
15 Command Line Tools... 99
15.1 Benerator... 99
15.2 DB Snapshot Tool... 99
15.3 XML Creator... 99
16 Domains... 101
16.1 person domain... 102
16.1.1 PersonGenerator... 102
16.1.2 PersonGenerator Properties... 103
16.1.3 Person Class... 103
16.1.4 Supported countries... 103
16.2 Address domain... 106
16.3 net domain... 107
16.4 organization domain... 107
16.5 finance domain... 109
16.6 product domain... 110
16.7 br domain... 110
16.8 us domain... 110
17 Component Reference... 111
17.1 Generators... 111
17.1.1 Domain Generators... 111
17.1.2 Common Id Generators... 111
17.1.3 Database-related Generators... 111
17.1.4 simple type generators... 111
17.1.5 current date / time generators... 111
17.1.6 arbitrary date / time generators... 112
17.1.7 file related generators... 112
17.1.8 State Generators... 112
17.1.9 Seed Based Generators... 112
17.2 Distributions... 113
17.2.1 Memory consumption... 113
17.2.2 Sequences... 113
17.2.3 CumulativeDistributionFunction... 116
17.2.4 ExponentialDensityIntegral... 116
17.2.5 Weight Functions... 116
17.2.6 GaussianFunction... 116
17.2.7 ExponentialFunction... 116
17.2.8 DiscreteFunction... 117
17.3 Converters... 118
17.3.1 Databene Converters... 118
17.3.2 Java Formats... 119
17.4 Validators... 120
17.4.1 Domain Validators... 120
17.4.2 Common validators... 120
17.4.3 Tasks... 120
17.5 Consumers... 121
17.5.1 LoggingConsumer... 121
17.5.2 ConsoleExporter... 121
17.5.3 JavaInvoker... 121
17.5.4 DbUnitEntityExporter... 122
17.5.5 XMLEntityExporter... 122
17.5.6 NoConsumer... 122
17.5.7 ScriptedEntityExporter... 123

17.5.8 FixedWidthEntityExporter.. 124
17.5.9 XLSEntityExporter.. 125
17.5.10 CSVEntityExporter.. 126
17.5.11 SQLEntityExporter.. 127
17.6 EntitySources (Importers).. 128
17.6.1 DbUnitEntitySource.. 128
17.6.2 CSVEntitySource.. 128
17.6.3 FixedColumnWidthEntitySource.. 128
17.6.4 XLSEntitySource.. 129
17.7 Benerator Utility Classes.. 129
17.7.1 RandomUtil.. 129
17.8 Databene Commons Library.. 129
17.8.1 TimeUtil.. 130
17.8.2 Period.. 131
17.8.3 IOUtil.. 131
17.8.4 CharUtil.. 131
18 Using DB Sanity.. 132
19 Maven Benerator Plugin.. 133
19.1 System Requirements.. 133
19.2 Getting started.. 133
19.3 Common configuration elements.. 134
19.4 Executing descriptor files.. 134
19.5 Creating database snapshots.. 134
19.6 Creating XML files from XML Schema files.. 135
19.7 Creating a project assembly.. 135
19.8 Extending the classpath.. 135
19.9 Profile-based configuration.. 136
19.10 Attaching the Mojo to the Build Lifecycle.. 137
20 Extending benerator.. 138
20.1 Custom Generators.. 138
20.1.1 Generator Interface.. 138
20.1.2 Generator States.. 139
20.1.3 Helper classes for custom Generators.. 139
20.2 Custom FreeMarker methods.. 140
20.3 Custom Sequences.. 141
20.4 Custom WeightFunctions.. 141
20.5 Custom CumulativeDistributionFunctions.. 141
20.6 Custom Converters.. 142
20.7 Custom Validators.. 143
20.7.1 javax.validation.ConstraintValidator.. 143
20.7.2 org.databene.commons.Validator.. 143
20.7.3 Implementing both interfaces.. 143
20.8 Custom Consumers.. 144
20.9 Custom EntitySources.. 144
20.10 Custom Tasks.. 144
21 Using Benerator as Load Generator.. 146
21.1 JavaInvoker.. 146
21.2 Checking performance requirements.. 146

22 Troubleshooting......148
22.1 (Out of) Memory......148
22.2 temp Directory......148
22.3 File Encoding......148
22.4 Logging......148
22.5 Locating Errors......149
22.6 Database Privilege Problems......149
22.7 Constraint Violations......150
22.8 'value too large for column' in Oracle......150
22.9 Time is cut off in Oracle dates......151
22.10 Composite Keys......151
22.11 Importing Excel Sheets......151
22.12 Number Generation on Oracle......151
22.13 Unknown Column Type......151
22.14 Table 'X' not found in the expected catalog 'Y' and schema 'Z'......151
23 Monitoring Benerator......152
23.1 Monitoring with JConsole......152
23.2 Remote monitoring......153
24 Benerator Performance Tuning......154
24.1 Performance......154
24.2 pageSize......154
24.3 JDBC batch......154
24.4 Query fetch size......154
24.5 Id Generation......154
24.6 Scripts......155
24.7 Parsing (Oracle) metadata......155
24.8 Distributed Execution......155

Preface

This document is supposed to become a complete summary of everything you need to learn benerator usage, use it efficiently and extend it as you need. This reference is under construction and will update it about every two weeks. So check back regularly for updates at the databene web site.

If problems remain unsolved after reading this book and the databene forum, do not hesitate to contact me, Volker Bergmann, for help. benerator is open source and provided for free. So if you need assistance, you are strongly welcome to buy consulting services and personal on-site project support.

Since you can do quite a lot of different things with benerator but surely are interested in just a part of it, here's some guidance:

'Introduction to benerator', introduces you to goals and features of benerator and advises you how to install a binary distribution and how to get the sources and set up an Eclipse project for using, debugging and customizing benerator.

'Data Generation Concepts', **'Descriptor File Format'** and **'Advanced Topics'** then provide you with a structured and complete introduction into the benerator descriptor file setup.

Benerator supports a multitude of service provider interfaces (SPIs). It comes along with some implementations for specific business domains (**'Domains'**) and general purpose classes in **'Component Reference'**. Finally you are instructed how to write custom SPI implementations in **'Extending benerator'**.

1 Introduction to benerator

1.1 Goals

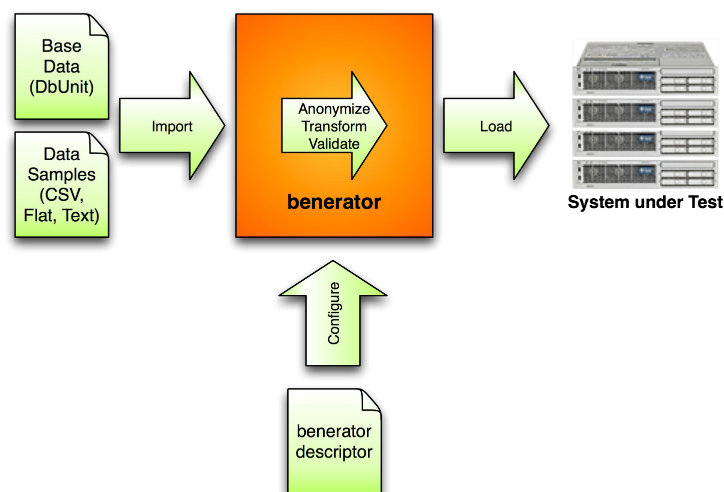
The core goals of benerator are

- Generation of data that satisfies complex data validity requirements
- Efficient generation of large data volumes
- Early applicability in projects
- Little maintenance effort with ongoing implementation through configuration by exception
- Wide and easy customizability
- Applicability by non-developers
- Intuitive data definition format
- Satisfying stochastic requirements on data
- Extraction and anonymization of production data
- Supporting distributed and heterogeneous applications
- Establishing a common data generation platform for different business domains and software systems

1.2 Features

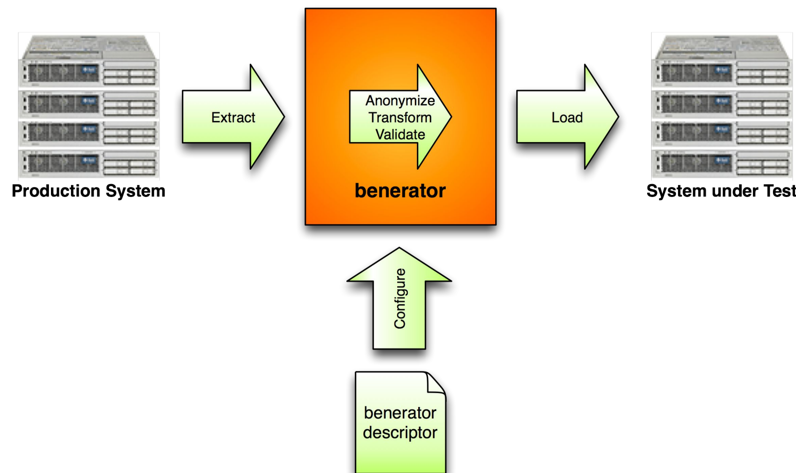
1.2.1 Data Synthesization

Performance test data can be completely synthesized. A basic setup can be imported e.g. from DbUnit files, CSV files and fixed column width files. A descriptor file configures how imported data should be processed and adds completely synthesized data. The processed or generated data finally is stored in the system under test.



1.2.2 Production Data Anonymization

Production data can be easily extracted from production systems. Tables can be imported unmodified, filtered, anonymized and converted.

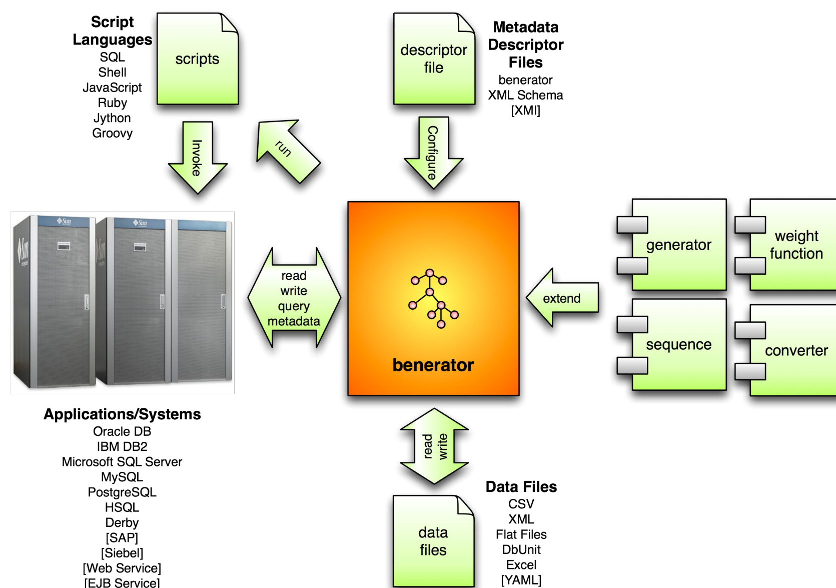


1.3 State of the benerator

benerator is developed and continuously extended and improved since June 2006. benerator is mainly used and tested best for data file and database data generation, for these applications benerator should help you for almost all your data generation needs out of the box - and extending benerator for specific needs is easy.

XML-Schema, on the other hand, allows for an extraordinarily wide range of features. benerator's XML support is limited to features that are useful for generating XML data structures (no mixed content) and does not yet support all variants possible with XML schema. The elements <unique>, <key> and <keyRef> cannot be handled automatically, but require manual configuration. The following features are not yet implemented: <group>, <import>, <all> and <sequence> with minCount != 1 or maxCount != 1. If you need support for some of these, please contact me.

1.4 Building Blocks



1.5 Database Support

All common SQL data types are supported (for a list of unsupported types, see ???)

benerator was tested with and provides examples for

- Oracle 10g (thin driver)
- DB2
- MS SQL Server
- MySQL 5
- PostgreSQL 8.2
- HSQL 1.8
- H2 1.2
- Derby 10.3
- Firebird

2 Installation

2.1 Download the distribution binary

Download benerator from http://sourceforge.net/projects/benerator/files/.

You should download the most recent version of the benerator-dist archive from the download page, e.g. databene-benerator-0.8.1-dist.zip.

2.2 Unzip Benerator

Unzip the downloaded file in an appropriate directory, e.g. /Developer/Applications or C:\Program Files\Development.

2.3 Set BENERATOR_HOME

Create an environment variable BENERATOR_HOME that points to the path you extracted benerator to.

Windows Details: Open the System Control Panel, choose Advanced Settings - Environment Variables. Choose New in the User Variables section. Enter BENERATOR_HOME as name and the path as value (e.g. C:\Program Files\Development\databene-benerator-0.8.1). Click OK several times.

Mac/Unix/Linux Details: Open the file .profile in your user directory. Add an entry that points to benerator, e.g.: export BENERATOR_HOME=/Developer/Applications/databene-benerator-0.8.1

2.4 Optional: Install JDBC drivers

Benerator comes with open source JDBC drivers (for connecting to a database). No extra installation is necessary for them:

- jTDS Driver (MS SQL Server or Sybase)
- MySQL Connector
- HSQL DB
- H2 DB
- Derby Client
- PostgreSQL
- Jaybird (Firebird DB)

However, if you need to use a closed source database driver, the vendor usually requires you to accept usage conditions before you can download and install their driver. So, if you are using Oracle DB or DB2, get the JDBC drivers from these sites:

- Oracle [http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html]: Click on "Oracle Database 11g Release 2 (11.2.0.1.0) drivers". Download ojdbc6.jar and put it into benerator's lib directory. You will need to create a free Oracle account.

- DB2 [http://www-306.ibm.com/software/data/db2/java]: Follow the link 'IBM Data Server Driver for JDBC and SQLJ', download the driver archive and copy the contained file db2jcc.jar to benerator's lib directory.

2.5 Optional: Set up log4j

Troubleshooting is simpler if you make use of log4j's configuration capabilities. If you do not know (or care) about logging, simply skip this step. Otherwise put a custom log4j.xml or log4j.properties file into the BENERATOR_HOME/lib directory.

2.6 On Unix/Linux/Mac systems: Set permissions

Open a shell on the installation's root directory and execute

```
chmod a+x bin/*.sh
```

2.7 Mac OS X configuration

On Mac OS X you need to provide benerator with an explicit configuration of the JAVA_HOME path. See http://developer.apple.com/qa/qa2001/qa1170.html for a good introduction to the OS X way of setting up Java. It is based on aliases conventions. If you are not familiar with that, you should read the article. If Java 6 (or newer) is the default version you will use, you can simply define JAVA_HOME by adding the following line to your .profile: in your user directory:

```
export JAVA_HOME=/Library/Java/Home
```

If it does not work or if you need to use different Java versions, it is easier to 'hard-code' JAVA_HOME like this:

```
export  JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home
```

2.8 Verifying the settings

On a Windows system, open a console window (cmd.exe) and type

benerator –version

On all other systems invoke the command line

benerator.sh –version

Benerator will then launch and print out version information about itself, the Java version it uses, the operating system and the installed Script engines, e.g.:

```
Benerator 0.8.1
Java version 1.6.0_22
JVM Java HotSpot(TM) 64-Bit Server VM 17.1-b03-307 (Apple Inc.)
OS Mac OS X 10.6.6 (x86_64)
Installed JSR 223 Script Engines:
- Mozilla Rhino[ejs, EmbeddedJavaScript, embeddedjavascript]
- Mozilla Rhino[rhino-nonjdk, js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
- AppleScriptEngine[AppleScriptEngine, AppleScript, OSA]
- Mozilla Rhino[js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
```

3 The Benerator Project Wizard

The Penerator Project Wizard helps you to easily and quickly set up any kind of Benerator project:

- Descriptor file-driven data generation
- XML Schema-driven XML file generation
- Creation of database snapshots

3.1 Starting the wizard

Start the Project Wizard on the command line by typing

[on Windows:]

```
benerator-wizard.bat
```

[or Mac/Unix/Linux/Solaris:]

```
benerator-wizard.sh
```

Depending on your language settings, the GUI welcomes you with an English or German dialog (contributions of translations are welcome). You can override the default language settings by using Java's user.language system property, e.g.

```
benerator-wizard.sh -Duser.language=en
```

The dialog looks like this:

3.2 Configuring the project

Enter the project name and choose an appropriate directory.

Then you can choose among a list of predefined project types:

- **Hello World example**: Prints 'Hello World'-style output to the console
- **Simple Project**: Simple project definition for custom generation projects
- **Generate CSV file**: Defines generation of a simple CSV file
- **Generate fixed column width file**: Defines generation of a simple fix column width file
- **Generate Excel(TM) document**: Defines generation of a simple Excel(TM) file
- **Populate database**: Small project that defines a database, creates a table and populates it
- **Reproduce & scale existing database**: Creates a snapshot of the current database content and creates a descriptor which replays the snapshot and defines templates for adding arbitrary volumes of data to each table.
- **Shop example database**: Performs Definition and population of the databene shop database example for all database systems supported by Benerator: DB2, Derby, Firebird, HSQL, H2, Oracle, PostgreSQL and SQLServer.
- **Generate simple XML file**: Defines generation of a simple XML file
- **Create XML documents from XML Schema**: Uses an (annotated) XML Schema file for generating several XML files
- **Create a database snapshot**: Configures database snapshot generation
- **Write Benerator extensions**: Java project that provides you with sample implementations of Benerator's extension interfaces and makes them cooperate in a descriptor file.

You can optionally make the wizard create an Eclipse project configuration, too. This requires you to have Maven istalled.

If necessary, specify the database connection settings and test if they are right by clicking 'Test Connection'.

When reproducing an existing database or creating a database snapshot, choose a snapshot file format: 'DBUnit file', 'Excel(TM) document', or 'SQL file'. In all other cases, choose 'None'.

When reproducing an existing database, you need to provide Benerator with the DDL files for creating and dropping the related tables and sequences ('create tables' script and 'drop tables' script).

Finally you can explicitly choose system dependent settings. If you leave these fields empty, Benerator will always take the individual settings of the system it is executed on.

3.3 Creating and running the project

Finally press 'Create' for creating the project. The wizard then configures the project in the specified project folder.

Look into the project folder and read the readme.txt file. This contains individual instructions for your project: What further configuration steps are eventually necessary (e.g. proprietary database drivers), how to invoke Benerator for this special project type, and how to go on with customizations.

The easiest way to run the generated project is to run the Maven Benerator Plugin as described in the generated readme.txt files. The project wizard creates all necessary files for you to execute the created projects immediately (except for proprietary database drivers: Oracle or DB2). However you can also invoke Benerator directly from the command line, but have to take care of compiling custom Java classes in the project for yourself.

4 Quick tour through the descriptor file format

4.1 <setup>

The benerator configuration file is XML based. An XML schema is provided. The document root is a setup element:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<setup xmlns="http://databene.org/benerator/0.8.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://databene.org/benerator/0.8.1
    http://databene.org/benerator-0.8.1.xsd">

  <!-- content here -->

</setup>
```

benerator descriptor files are supposed to be named *'benerator.xml'* or end with the suffix *'.ben.xml'*.

4.2 benerator properties

Several global benerator properties allow for customization of its behavior:

name	description	default setting
defaultEncoding	the default file encoding to use for reading and writing text files	the system's file encoding
defaultLineSeparator	the line separator to use by default	the system's line separator
defaultTimeZone	The time zone to use	The system's time zone
defaultLocale	The locale to use if none has been specified explicitly	The system's language code, e.g. 'de'
defaultDataset	The dataset to use if none has been specified explicitly	The system's country's two-letter ISO code, e.g. 'US'
defaultPageSize	the number of entities to create in one 'run', typically a transaction	1
defaultScript	The default script engine to use for evaluating script expressions	ben (DatabeneScript)
defaultNull	tells if nullable attribute should always be generated as null by default	true
defaultSeparator	the default column separator to use for csv files	,
defaultErrorHandler	the default error handling mechanism to use	fatal
validate	Boolean flag to turn off validation (e.g. of XML validity and type definition consistency).	true
maxCount	limits the maximum cardinality of all entity and association generations. If set to 0, cardinalities will not be limited.	-1
defaultOneToOne	When set to to Benerator assumes each relation is one-to-one.	false
acceptUnknown	When set to true, Benerator accepts unknown simple data types	false

SimpleTypes	from its DescriptorProviders, relying on the user to choose the correct data type when generating.	

You can configure them in the <setup> element, e.g.

```
<setup xmlns=...
  defaultEncoding="ISO-8859-1"
  defaultPageSize="1000">
```

4.3 <include>

4.3.1 Inclusion of properties files

An alternative way to specify the Benerator properties from the previous chapter is to specify them in a properties file, e.g.

```
context.defaultEncoding=UTF-8
context.defaultPageSize=1000
```

and include the properties file in the benerator descriptor file:

```
<include uri="my.properties"/>
```

This way you can easily use different settings in different environments (see Section 4.7, "Staging").

File entries that do not begin with 'benerator" are simply put into the generation context and can be used to configure generation behavior.

4.3.2 Sub-Invocation of descriptor files

Besides properties files, Benerator descriptor files can be included too, e.g.

```
<include uri="subgeneration.ben.xml" />
```

4.4 Global settings

benerator supports global settings. They can be evaluated using script expressions, e.g. {user_count}. This way, different types of settings may be evaluated:

- system environment
- Java virtual machine parameters
- context variables

A setting is explicitely defined using a setting element:

```
<setting name="threshold" value="5"/>
```

4.5 <import>

Benerator has lots of plugin interfaces, but is agnostic of most implementors. So you need to explicitly import what you need.

The following packages are imported by default (providing, for example, the ConsoleExporter):

org.databene.benerator.consumer	General-purpose consumer classes
org.databene.benerator.primitive	Generators for primitive data types
org.databene.benerator.primitive.datetime	Generators for date, time and timestamp data
org.databene.benerator.distribution.sequence	Distributions of 'Sequence' type

org.databene.benerator.distribution.function	Distributions of 'Function' type
org.databene.benerator.distribution.cumulative	Distributions of type 'CumulativeDistributionFunction'
org.databene.benerator.sample	Generator components that use sample sets or seeds
org.databene.model.consumer	ConsoleExporter and LoggingConsumer
org.databene.commons.converter	Converter components from databene-commons
org.databene.commons.format	Format components from databene-commons
org.databene.commons.validator	Validator components from databene-commons
org.databene.platform.fixedwidth	Fixed column width file importer and exporter
org.databene.platform.csv	CSV file importer and exporter
org.databene.platform.dbunit	DbUnit file importer and exporter
org.databene.platform.xls	Excel(TM) Sheet importer and exporter

Benerator extensions can be bundled as domains (logical extensions) or platforms (technical extensions). You can export different bundles as comma-separated lists:

```
<import domains="address, net" />
<import domains="organization" />
<import platforms="csv, db" />
```

Imports must be the first elements used in a descriptor file.

When using a Benerator plugin or another library, you need to make sure that Benerator finds its binary. There are three alternatives:

1. Putting the associated jar file(s) into the lib folder of your Benerator installation. This way it is avaliable for all data generation projects on your machine. If you work in a team where everyone in familiar with Benerator and the toolset is not based on Maven, this is generally the preferred approach.
2. Create a sub folder named lib under the data generation project folder and put the jar file(s) there. When distributing the project to be executed on machines with plain Benerator installations, distribute the full folder content including the lib sub folder.
3. When using Maven to run Benerator, simply create the necessary Maven dependencies and Maven will acquire all needed libraries dynamically. Read more about this in the chapter 'Maven Benerator Plugin'.

4.6 <generate>

<generate> elements are used to generate data from scratch. There are lots of configuration options. The minimal configuration specifies the type of data to be generated. For now, all generated data are 'entities' (composite data).

```
<generate type="Person" count="10" consumer="ConsoleExporter"/>
```

This will make Benerator generate 10 'Person' Entities and send it to a ConsoleExporter that prints out the persons to the console. But what is a Person? Benerator will figure it out by itself, if it knows e.g. a database with a 'PERSON' table, an XML schema with a 'Person' element or any other 'DescriptorProvider'. Benerator will generate database-valid or XML-Schema-valid data automatically. More about this later.

Let us start without DescriptorProviders, manually putting together what we need.

Entities consist of members, e.g. **<attribute>**s, **<id>**s or **<reference>**s. I will concentrate on attributes in the following sections and explain ids and references later.

4.6.1 "constant"

The simplest way to define data generation is using the same value for all generated data:

```
<generate type="Person" count="10" consumer="ConsoleExporter">
    <attribute name="active" type="boolean" constant="true" />
</generate>
```

So we define, that all Person entities are generated with an 'active' attribute of type 'boolean' that is set to 'true'.

4.6.2 "values"

Attributes may be randomly set from a list of comma-separated values

```
<generate type="Person" count="10" consumer="ConsoleExporter">
    <attribute name="firstName" type="string" values="'Alice','Bob','Charly'" />
    <attribute name="rank" type="int" values="1,2,3" />
</generate>
```

So we define, that Person entities have a 'firstName' attribute that is 'Alice', 'Bob' or 'Charly' and a rank of 1, 2 or 3. Note that string literals must be 'quoted', while number or Boolean literals do not.

4.6.3 "pattern": Generation by Regular Expression

String attribute generation can be configured using the "pattern" attribute with a regular expression, for example:

```
<generate type="Person" count="10" consumer="ConsoleExporter">
    <attribute name="salutation" type="string" pattern="(Mr|Mrs)" />
    <attribute name="postalCode" type="string" pattern="[1-9][0-9]{4}" />
</generate>
```

You can find a detailed descripton about Benerator's regular expression support in the chapter „Regular Expression Support".

4.7 <iterate>

The <iterate> element is used to iterate through pre-existing data, e.g. in a data file or database. The general form is

```
<iterate type="Person" source="persons.csv" />
```

which iterates through all Persons defined in a CSV-file called 'persons.csv'.

By default, iteration goes once from beginning to the end, but later you will learn about many ways to iterate repeatedly, apply distributions or filter the data to iterate through.

4.8 "offset"

In whatever type of data generation or iteration, an **offset** can applied to skip the heading entries of a data source, e.g.

```
<iterate type="Person" source="persons.csv" offset="10" />
```

leaves out the first ten entries of the persons.csv file.

4.9 <echo>

The meaning of the <echo> element is similar to the echo command in batch files: Simply writing information to the console to inform the user what is happening, e.g.

```
<echo>Running...</echo>
```

For Mac OS X users there is a nice extra feature: When using **type='speech'**, Benerator uses Mac OS X's speech facility to speak the text. When executed on other operating systems, the text is only printed to the console:

```
<echo type='speech'>Generation Finished</echo>
```

4.10 <beep/>

makes Benerator emit a short beep

4.11 <comment>

The <comment> element also prints output, not to the console, but to a logger. Thus you have the option of configuring whether to ignore the output or where to send it to.

```
<comment>Here we reach the critical part...</comment>
```

Using XML comments <!-- --> instead of comment descriptors would make it harder for you to comment out larger portions of a file for testing and debugging.

4.12 <execute type="shell">

The <execute> element serves to execute different kinds of code. One option is the execution of shell commands:

```
<execute type="shell">start-database.sh</execute>
```

The program output is printed to the console.

Note that some windows shell commands are only available in the command line interpreter. In order to invoke them, you need to call cmd /C, e.g.

```
<execute type="shell">cmd /C type myGeneratedFile.csv</execute>
```

You can use <execute> for invoking scripts too (SQL, DatabeneScript, JavaScript, FreeMarker and more), but that will be explained later.

4.13 <wait>

The <wait> element makes Benerator wait for a fixed or a random amount of time.

A fixed amount of time is useful, e.g. for waiting until a system is initialized:

```
<wait duration="20000" />
```

The duration is the time in milliseconds.

Random periods of wait time are useful when using Benerator to simulate client activity on a system. For this, you can nest <wait> elements in <generate> elements. More about this later.

4.14 <error>

You can make Benerator signal an error with a message and code:

```
<error code="-3">An error has occured</error>
```

If Benerator is not configured to do otherwise, it prints out the error message, cancels execution, finishes the process and returns the exit code to the operating system. If no exit code is specified, Benerator uses -1.

4.15 <if>

Evaluates a script expression and executes sub elements depending on the result.

Either a decision to execute something or not:

```
<if test="org.databene.commons.SystemInfo.isWindows()">
    <echo>Running under Windows</echo>
</if>
```

or a decision between alternatives:

```
<if test="org.databene.commons.SystemInfo.isWindows()">
    <then>
        <execute type="shell">cmd /C type export.csv</execute>
    </then>
    <else>
        <execute type="shell">cat export.csv</execute>
    </else>
</if>
```

A typical application of the <if> element is to check if a required configuration is defined, and if not, to fall back to a default...:

```
<if test="!context.contains("stage")">
    <echo>No stage defined, falling back to 'dev'</echo>
    <setting name="stage" value="dev" />
</if>
```

...or to report an error:

```
<if test="org.databene.commons.SystemInfo.isWindows()">
    <error>No stage has been set</error>
</if>
```

4.16 <while>

The <while> element executes sub elements as long as a boolean 'test' expression resolves to **true**:

```
<while test="!target.isAvailable()">
    <wait duration="500" />
</while>
```

4.17 <id> - Generating unique identifiers

For marking an entity member as identifier, it is declared with an <id> element, e.g.

```
<id name="identifier" type="long"/>
```

There are several special id generators available. If you do not specify one explicitly, Benerator takes the IncrementalIdGenerator.

For explicitly choosing or initializing an id generator, use the generator attribute, e.g.:

```
<id name="identifier" type="long" generator="new IncrementalIdGenerator(100)" />
```

for using an IncrementalIdGenerator, that starts with the value 100.

See the chapter „Common ID Generators" for a complete ID generator reference and „Using databases" for database-related id generators.

Instead of using a generator, you can as well use other <attribute>-like features, e.g. scripts:

```
<id name="id" type="long" script="parent.id" />
```

4.18 Naming Conventions

For automatic support of special file content, the following naming conventions apply:

File Name	File Type
*.ben.xml	benerator descriptor file
*.dbunit.xml	DbUnit data field
*.csv	CSV file with data of simple type
*.ent.csv	CSV file with entity data
*.wgt.csv	CSV file with weighted data of simple type
*.fcw	Fixed column width files with entity data
*.set.properties	Dataset nesting definition

5 Data Generation Concepts

Now that you have mastered the first tutorials and have had a glance on benerator's features, it is time for an in-depth introduction to data generation:

5.1 Naming

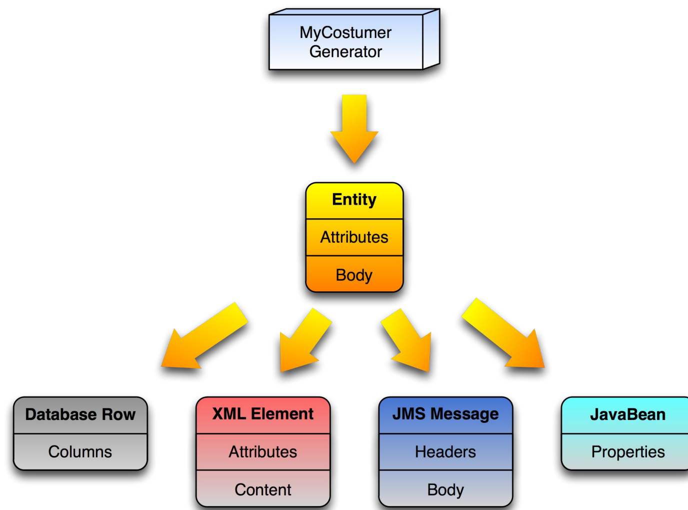
Business objects are called *entity* in this book, their contained simple type data are *attributes* .

5.2 Entity Data

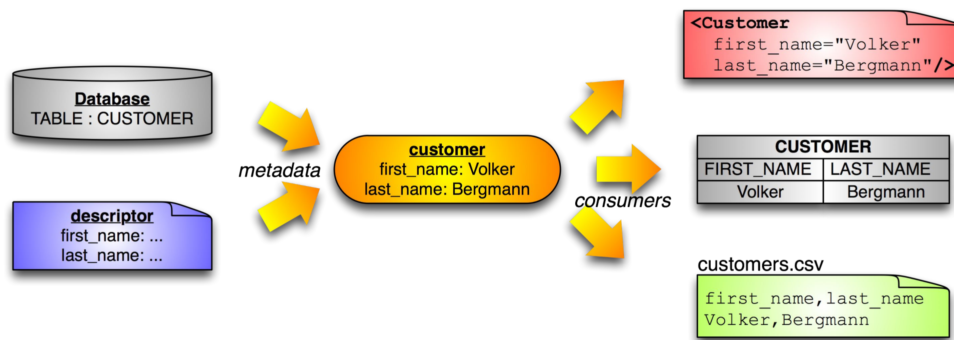
benerator generates entities in a platform-independent manner (internally using the class org.databene.model.data.Entity. An entity will be interpreted individually depending on the target system. It can be mapped to

- relational data (DB)
- hierarchical data (XML)
- graphs (JavaBeans)
- attributed payload holders (File, JMS Message, HTTP invocations)

So you can use an abstract, generic way of defining and generating business objects and reuse it among colleagues, companies and target platforms. An address generator defined once can be used for populating customer tables in a database or creating XML order import batch files.



Metadata is platform-neutral too. So benerator can import metadata definitions from a database and use it for generating XML data:



Entities can have

- an arbitrary number of simple-type attributes (like database tables or XML attributes). They can have a cardinality > 1, too (like arrays of simple types)
- sub components of entity type (like XML sub elements)
- a body of simple type (like XML simpleType elements, files or JMS messages)

5.3 Simple Data Types

benerator abstracts simple types too. These are the predefined simple types:

benerator type	JDBC type name	JDBC type code	Java type
byte	Types.BIT	-7	java.lang.Byte
byte	Types.TINYINT	-6	java.lang.Byte
short	Types.SMALLINT	5	java.lang.Short
int	Types.INTEGER	-5	java.lang.Integer
big_integer	Types.BIGINT	-5	java.lang.Long

float	Types.FLOAT	6	java.lang.Float
double	Types.DOUBLE	8	java.lang.Double
double	Types.NUMERIC	2	java.lang.Double
double	Types.REAL	7	java.lang.Double
big_decimal	Types.DECIMAL	3	java.math.BigDecimal
boolean	Types.BOOLEAN	16	java.lang.Boolean
char	Types.CHAR	1	java.lang.Character
date	Types.DATE	91	java.lang.Date
date	Types.TIME	92	java.lang.Date
timestamp	Types.TIMESTAMP	93	java.lang.Timestamp
string	Types.VARCHAR	12	java.lang.String
string	Types.LONGVARCHAR	-1	java.lang.String
string	Types.CLOB	2005	java.lang.String
object	Types.JAVA_OBJECT	2000	java.lang.Object
binary	Types.BINARY	-2	byte[]
binary	Types.VARBINARY	-3	byte[]
binary	Types.VARBINARY	-4	byte[]
binary	Types.BLOB	2004	byte[]
(heuristic)	Types.OTHER	1111	(heuristic)

Oracle's NCHAR, NVARCHAR2 and NCLOB types are treated like strings.

The following JDBC types are not supported: DATALINK (70), NULL (0), DISTINCT (2001), STRUCT (2002), ARRAY (2003), REF (2006). If you need them, tell me (volker@databene.org).

5.4 Data Characteristics

5.4.1 Distribution Concept

There are two special issues which often remain unaddressed in testing:

- using realistic probability distributions (e.g. popularity of shop items)
- creating unique values (e.g. IDs or unique phone numbers for fraud checking)

For these purposes, databene benerator provides several interfaces, which extend a common interface,

Distribution. The most important ones are

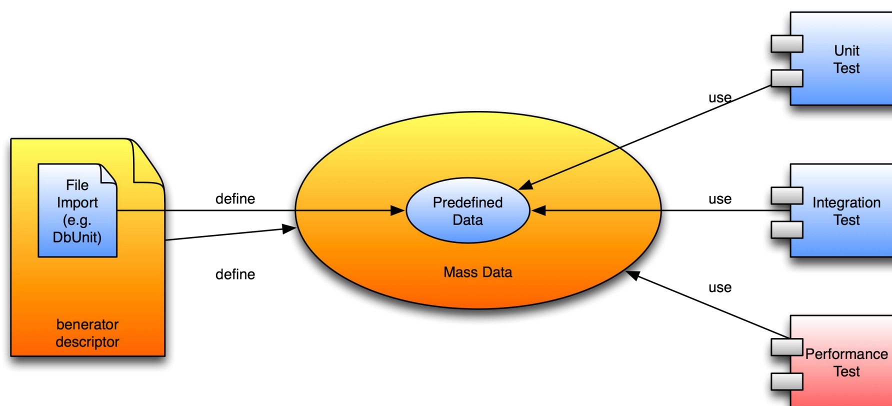
- WeightFunction
- Sequence

For a list of predefined distributions, see Section 8.2, “Distributions”.

5.5 Generation Stages

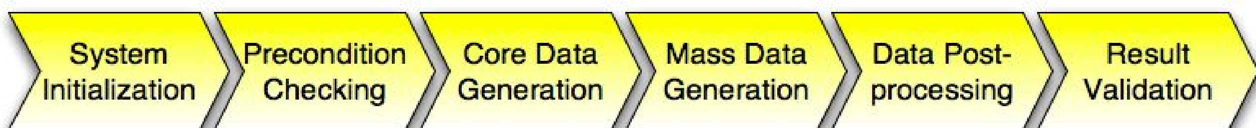
The result of data generation typically consists of

- a small predefined and well-known core data set
- large data volumes that are generated randomly and extends the core data



This approach has the advantage of supporting different test types with the same generation setup: It is essential for performance tests to have a unit- and integration-tested system and you can strongly simplify the testing procedure by reusing data definitions from unit- and integration tests as core data for a mass data generation.

Regarding the technical steps involved, a generation process employs up to six stages for each system involved:



- **System Initialization**, e.g. by start scripts and DDL scripts
- **Precondition Checking** for verifying that data required for data generation is available
- **Core Data Generation** for creating a predefined data set
- **Mass Data Generation** for scaling data amounts of large volume
- **Data Postprocessing** for performing complex operations
- **Result Validation** for verifying the generated data

5.5.1 System Initialization Stage

In the system initialization stage you typically use scripts for starting and initializing the systems involved, e.g.

- start database by shell script
- run SQL script
- start application server

For starting a database with a shell script and initializing it with a SQL script, you could write:

```
<execute type="shell">sh ./startdb.sh &</execute>

<execute target="db" type="sql" onError="warn">
    DROP TABLE db_user;
    CREATE TABLE db_user (
        id       int         NOT NULL,
        name     varchar(30) NOT NULL,
        PRIMARY KEY  (id),
    );
</execute>
```

As you see, scripts can be inlined or imported from files. See Chapter 5, Scripting for a full introduction.

5.5.2 Precondition Checking Stage

Complex data generation is often split up int several stages of which each has its own preconditions. For example, if you want to generate orders for all kinds of products, you may want to assure that at least one product of each category is defined in the system.

The simplest way to perform precondition checks is the <evaluate> element, e.g. checking for categories without product:

```
<evaluate assert="{js:result == 0}" target="db">
   select count(*) from db_category
      left join db_product on db_product.category_id = db_category.id
      where db_product.category_id is null
</evaluate>
```

The <evaluate> element works as follows: First it evaluates a script the same way like a <execute> element does – In this example, the 'select' query is performed on the database. Then the result into a variable named 'result' and the 'assert' condition is evaluated which checks the value of the result and returns true or false. if the assertion resolves to 'false', benerator raises an error.

In this example, an error is raised if there is a category without any product assigned.

You can use an arbitrary expression language for performing the check. Like in <execute>, a prefix with colon can be used to indicate the script language. You can optionally add an 'id' attribute which will make Benerator put the evaluation result into the context with this id.

You can also call DB Sanity for verifying the preconditions, see the chapter about DB Sanity.

5.5.3 Core Data Generation Stage

For predefined data generation it is most convenient to import core data from a file - this gives you full control and easy configuration by an editor and is the most reliable source for reproducible data. Currently, the most convenient file formats for this task are DbUnit XML files (one file with several tables) and CSV (one file per table).

```
<!-- import integration test data for all tables from one DbUnit file -->
<iterate source="core.dbunit.xml" consumer="db" />


<!-- import predefined products from a CSV file -->
<iterate type="db_product" source="demo/shop/products.import.csv"
    encoding="utf-8" consumer="db" />
```

Fixed column width files and SQL files can be used too. If you need to import data of other formats you can easily write a parser and use it directly from benerator (See Section 9.9, “Custom EntitySources”).

5.5.4 Mass Data Generation Stage

Mass data generation is the primary goal of benerator and the main topic of this book. It is mainly performed by <generate> descriptors, which describe the creation of synthetic data, but may also include the import and reuse of information from external sources by an <iterate> descriptor. See the chapters *Chapter 3, Descriptor File Format* and *Chapter 4, Advanced Topics* for a description.

5.5.5 Data Postprocessing Stage

If your system has complex business logic (typically workflows), you will encounter generation requirements that are easier to satisfy by calling application business logic than by a pure descriptor-based generation.

For example you might need to generate performance test data for a mortgage application: People may apply for a mortgage, enter information about their house, incomes and expenses, their application is rated by some rule set and the mortgages finally is granted or rejected. The you have complex logic (rating) that is not necessarily useful to be reproduced for data generation. It is easiest to call the business logic directly.

This can be done in two ways:

- Scripts : Having script commands inlined in the benerator descriptor file or called from external files, e.g. DatabeneScript, JavaScript, Groovy, Ruby, Python. See Chapter 5, Scripting
- Tasks : Programming own Java modules that are invoked by benerator. See Section 4.6, “Tasks”

5.5.6 Result Validation Stage

Data generation may become quite tricky. For improving maintainability it is recommended to perform validations after data generation:

- Checking the number of generated objects
- Checking invariants
- Checking prerequisites for specific performance tests

You can do so with the <evaluate/> element, e.g. checking the number of generated customers

```
<evaluate assert="{js:result = 5000000}"
    target="db">select count(*) from db_user</evaluate>
```

The <evaluate> element was described above in the 'Precondition Checking' section.

You can also use DB Sanity for verifying the preconditions; see the chapter about DB Sanity.

5.6 Metadata Concepts

benerator processes metadata descriptors that can be imported from systems like databases and can be overwritten manually. Benerator automatically generates data that matches the (e.g. database) constraints. So, when it encounters a table defined like this:

```
CREATE TABLE db_user (
    ...
    active SMALLINT DEFAULT 1 NOT NULL,
    ...
);
```

When generating data for the user table, benerator will automatically generate all users with active set to 1:

```
<generate type="db_user" count="100" consumer="db"/>
```

If you specify active as an attribute, you inherit a new setting from the parent descriptor, dropping the parent's configuration of values=1 and adding a new one, e.g. the configuration

```
<generate type="db_user" count="100" consumer="db">
    <attribute name="active" values="0,1"/>
<generate>
```

will cause generation of 50% 0 and 50% 1 values.

5.6.1 Case Sensitivity

benerator has a heuristic case-sensitivity: It needs to combine metadata from different types of systems of which some may be case-sensitive, some may not. So benerator first assumes case-sensitivity when looking for a type. If the type is found in the same capitalization, this information used. If it is not found, benerator falls back to searching the type in a case-insensitive manner.

5.6.2 Namespaces

benerator has a heuristic namespace support, similar to case-sensitivity handling: when looking up a descriptor by name, benerator first searches the name in its assigned namespace. If the type is found there, this information used. If it is not found, benerator falls back to searching the type in all available namespaces.

5.6.3 <setting> and benerator identifiers

A benerator identifier (variable, entity or bean name) may contain only ASCII letters, numbers and underscores (no dot !) and is defined using a <setting> element - either in the descriptor file, e.g.

```
<setting name="user_count" value="1000000"/>
```

or in a properties file, e.g. myproject.properties:

```
user_count="1000000
```

which is then included in the descriptor file:

```
<include uri="myproject.properties"/>
```

Of course you can evaluate variables for defining other variables as well by using a script expression:

```
<setting name="event_count" value="{user_count * 10}"/>
```

A property can also refer to another element of the generation context:

```
<setting name="limit" ref="maxCount"/>
```

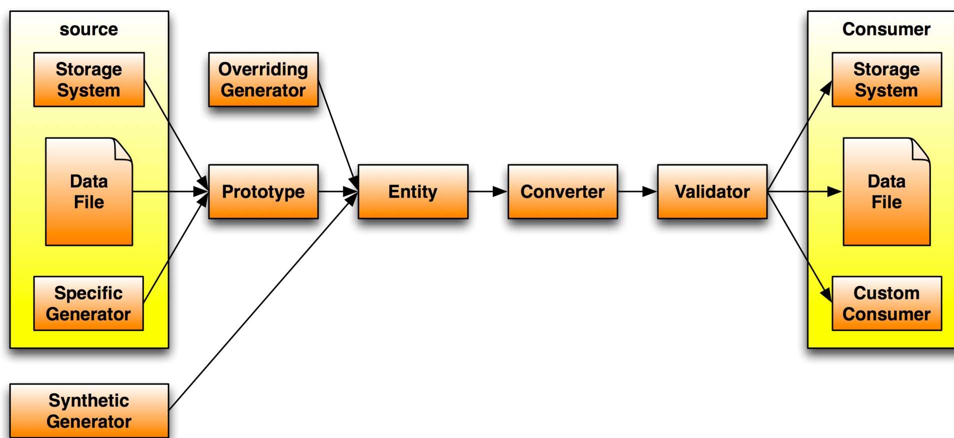
And it can be set calling a generator object:

```
<setting name="fileNumber" source="new DBSequenceGenerator('my_seq', db)"/>
```

You can define default values for properties. If no property value has been defined before, the property is set to this value:

```
<setting name="stage" default="development" />
```

5.7 Benerator Components



Entities as well as their attributes can be imported from storage systems , data files or specific generators . They then serve as prototypes of which attributes may be overwritten by another generator (overriding generator), e.g. for anonymization.

Alternatively, entities may be generated completely synthetically .

Entities and each entity attribute can be converted by a specific Converter object.

Validators assure validity of the generated entities and attributes. All entities that fail validation are discarded.

Finally, generated data is consumed by storing it in a storage system (e.g. database), writing it to a data file or using it in a custom Consumer implementation.

5.8 Instantiating Global Components

You ca define global components in a Spring-like syntax:

```
<bean id="helper" class="com.my.Helper">
  <property name="min" value="5"/>
  <property name="max" value="23"/>
</bean>
```

For details on this syntax and other variants, see the section “JavaBeans and the Benerator Context”. You can refer to such an object by its id ('helper' in this case).

5.9 Instantiating Local Components

The following chapters will introduce you to the usage of each component type available in benerator. They have common styles of definition and referral. If a component needs to be reused in different places, you would create it with a <bean> element and apply referral to use it. If you do not need to reuse one component in different places, there are more concise inline instantiation styles available:

- default construction
- parameterized construction
- property-based construction

5.9.1 Referral

Any class can be instantiated and made available to benerator by using a bean element, e.g. the 'helper' instance above, you can use it like this:

```
<attribute name="number" generator="helper"/>
```

This is called referral.

5.9.2 Default Construction

If you specify just a class name, benerator will create an instance of the class by invoking the default constructor. Be aware that the class needs a public no-argument constructor for being instantiated this way:

```
<attribute name="number" generator="com.my.Helper"/>
```

5.9.3 Parameterized Construction

You can as well specify the 'new' keyword, a class name and constructor parameters. benerator will then search a constructor with matching parameters and invoke it. If the class has several constructors with the same number of parameters benerator might choose the wrong one, so it is good practice to have just one constructor for each possible number of parameters.

```
<attribute name="number" generator="new com.my.Helper(5, 23)"/>
```

5.9.4 Property-based Construction

This is the most elegant and maintainable inline construction style, you specify the 'new' keyword, the class name and, in square brackets, a comma-separated list of name-value pairs for each JavaBean property. benerator uses a default constructor and the corresponding set...() methods to initialize the object.

```
<attribute name="number" generator="new com.my.Helper{min=5, max=23}"/>
```

5.10 Descriptive Data Generation

Descriptive Data Generation means defining elementary data restrictions , e.g. nullability and string lengths. For example, an attribute may have only one of an enumeration of values. They can be defined as a comma-separated list:

```
<attribute name="issuer" values="'AMEX','VISA'" />
```

For a list of descriptive attribute metadata, see Section 3.38, “Attribute Metadata Reference” Descriptive metadata can be imported automatically from database schema metadata and be used for automatic database-valid data generation:

5.11 Default Data Generation

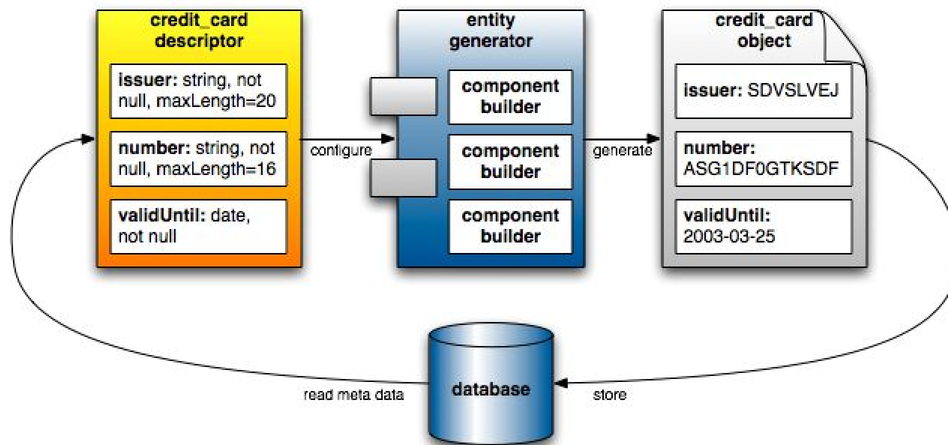
Based on descriptive metadata, benerator applies several defaults for generating database-valid data.

All nullable attributes are generated as null by default.

Primary keys are generated as integral numbers by default, starting from 1 and increased by 1 consecutively. Primary keys of string type are handled similarly.

Foreign keys are resolved automatically. For avoiding illegal generation cases, benerator assumes any foreign key relation to be one-to-one by default. Many-to-one relationships need to be configured manually.

Now have a look at an example for generating credit cards in a database:




```
<generate type="credit_card" count="500000" consumer="db"/>
```

benerator reads the metadata for the table credit_card from a database. This results in descriptive metadata , saying that a credit_card entity has three attributes: issuer and number of type string and validUntil of type date. All of them may not be null and the issuer attribute has a maximum length of 20 characters, the number of 16 characters.

This is enough information to make benerator generate, e.g. 50000 credit cards with a trivial setup:

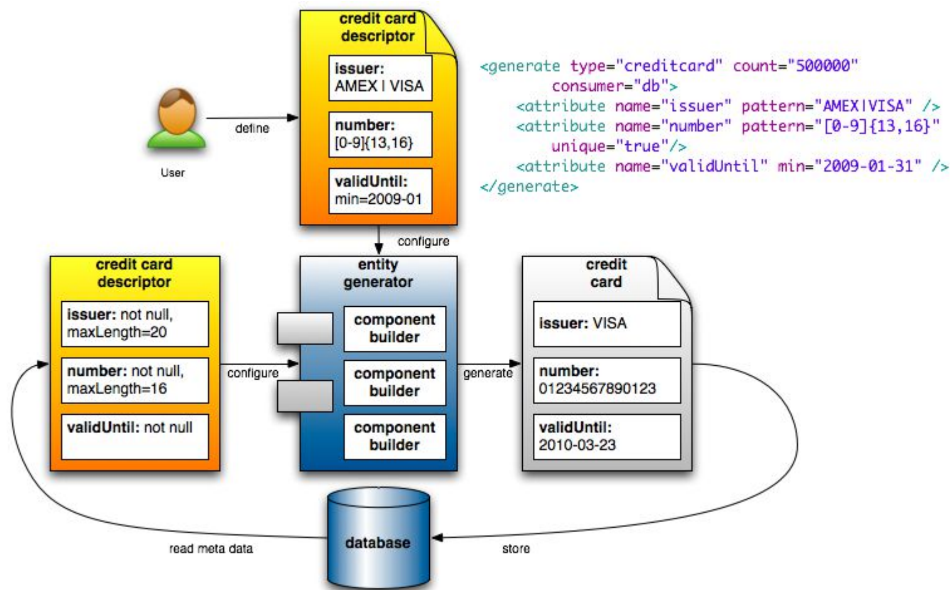
```
<generate type="credit_card" count="500000" consumer="db"/>
```

The resulting entries are database-valid automatically.

5.12 Constructive Data Generation

Constructive metadata describes methods of data generation, e.g. import from a data source or stochastic number generation. For a complete list of options, see ???.

We can improve the credit card example from above by adding own, constructive metadata to the descriptive ones imported from the database:

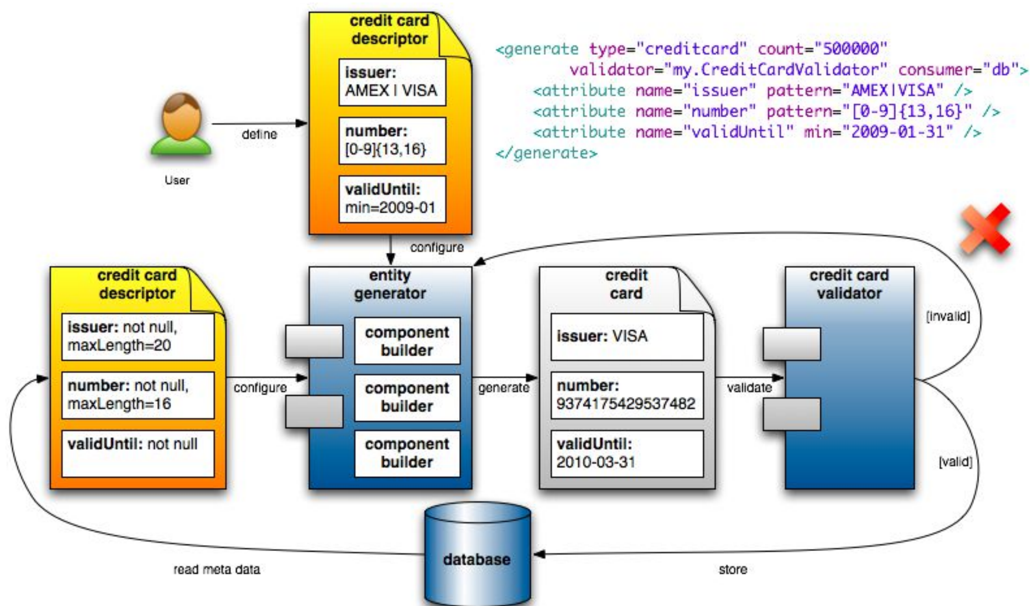


This way we can already satisfy simple validation algorithms, but not yet sophisticated ones that performs a checksum validation.

For a complete reference of metadata configuration, see Section 3.38, “Attribute Metadata Reference”, Section 3.25, “Generating IDs” and Section 3.26, “Resolving Relations”

5.13 Validating Data Generation

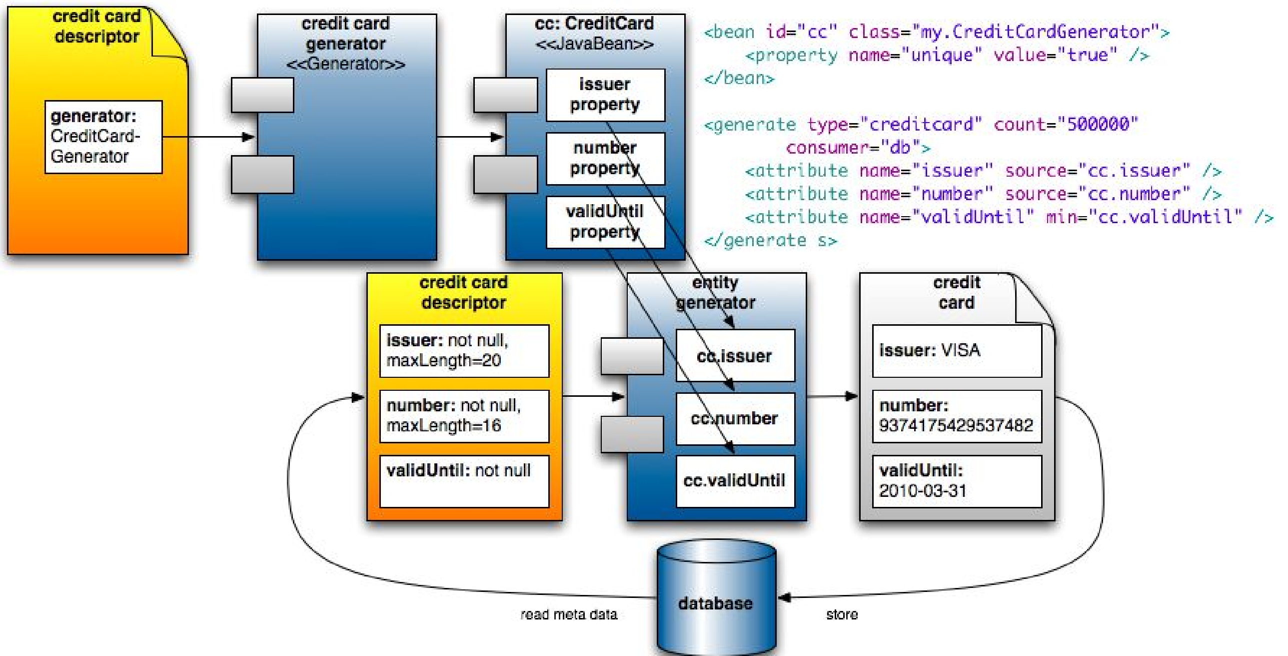
Suppose you have a validation component available, but do not know all details necessary for constructing valid data. In such a case, you can set up a constructive data generation and combine it with the validation module. So take the setup from the chapter before, write an adapter to your validation component and include it in benerator's data generation:



Your setup will then create random credit card setups and the credit card validator will discard the invalid ones. For the definition of custom validators, see Section 9.7, “Custom Validators”.

5.14 Prototype-based Data Generation

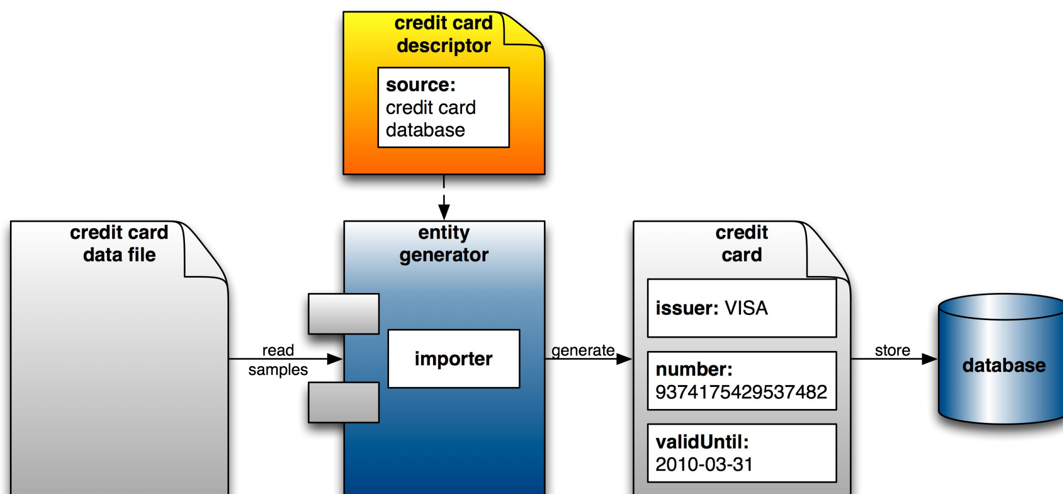
The examples above are satisfactory for almost all cases, but if you need to satisfy very difficult validity conditions you need ultimate control over generation. For our credit card example, you might have a validator module that connects to the credit card company and validates if the account really exists



You can generate prototypes with custom generators or import them as samples (See Section 3.11, “Sample-based Data Generation”).

5.15 Sample-based Data Generation

The examples above are satisfactory for almost all cases, but there are cases in which you need to use a predefined set of entities. For our credit card example, the tested application might check credit cards by connecting to the credit card company and query if the account really exists. In such a case you typically define a file with known credit card numbers to use:



```
<iterate type="credit_card"
    source="credit_cards.csv" consumer="db" />
```

You can use different types of data sources for templates:

- Files : CSV, fixed column width files, DbUnit. For importing data of custom file formats or from other sources, see Section 9.9, “Custom EntitySources”
- Storage Systems: Relational databases For importing data of proprietary storage systems, see Section 9.12, “Custom StorageSystems”

5.16 Variables

When importing entities from a data source you will need to map data in some way. This is where the variable concept comes in: You can define a <variable> as an auxiliary generator inside a <generate> descriptor and assign it a name (e.g. 'person'). In each entity generation this generator will provide a new generated object under the assigned name ('person'). So, if you want to access a part of a composite generated object you can query it e.g. by a script expression like person.familyName:

```
<generate type="customer" consumer="ConsoleExporter">

      <variable name="person" generator="PersonGenerator" />

      <attribute name="lastName" script="person.familyName" />

</generate>
```

For defining a variable, you can use the same syntax elements like for an attribute. But the type of data that the variable can generate is much less restricted. A variable may

- use an EntitySource or Generator that creates entity objects.
- use a Generator that creates Maps or JavaBean objects. Their map values or bean properties can be queried from a script the same way as for an entity
- execute a SQL query (e.g. name="c_customer" source="db" selector="select id, name from customer where rating = 0") of which column values may be accessed by a script (e.g. script="{c_customer[0]}" for the id).

5.17 Combining components and variables

Starting with Benerator 0.7, sub elements of a <generate> loop are evaluated in the order in which they appear in the descriptor file, in earlier versions they were reordered before processing. When nesting <generate> loops be aware, that each instance of the outer loop is consumed before a sub-generate is called, so it does not make sense to define an <attribute>, <id> or <reference> after the sub-generate statement.

5.18 Referring Files

In most cases, Files are referred by URIs. A URI may be

- a simple local (data.csv) or
- an absolute filename (C:\datagen\data.csv) or a
- a URL (http://my.com/datagen/data.csv).

For FTP access, use RFC1738 for encoding user name, password and file format, e.g. ftp://user:password@server/dir/file;type=i

5.19 Protocols

Currently benerator supports only file URIs for reading and writing and HTTP and FTP URIs for reading. Support of further protocols is possible and planned for future releases.

5.20 Relative URIs

Relative URIs are resolved in a HTML hypertext manner: A relative URL is interpreted relative to a 'base URI' which is the path of the benerator descriptor file. If file lookup fails, benerator searches the file relative to the current directory. If that fails to, benerator tries to retrieve the file from the Java classpath.

Benerator recognizes absolute paths under Windows (e.g. C:\test) and Unix (/test or ~/test). When in doubt, mark the URL as file URL: file:///C:/test or file:///test.

5.21 Importing Entities

Entities can be imported from 'system's, files or other generators. A typical application is to (re)use a DBUnit setup file from your (hopefully existing ;-) unit tests:

```
<iterate source="shop/shop.dbunit.xml" consumer="db"/>
```

For importing DbUnit files, follow the naming conventions using the suffix *.dbunit.xml*.

Each created entity is forwarded to one or more consumers, which usually will persist objects in a file or system, but might also be used to post-process created entities. The specified object needs to implement the Consumer or the system interface. When specifying a system here, it will be used to store the entities. File exporters (for CSV and fixed column width files) implement the Consumer interface.

5.22 Custom Importers

New import formats can be supported by implementing the EntitySource interface with a JavaBean implementation, instantiating it as bean and referring it by its id with a 'source' attribute, e.g.

```
<bean id="products_file"
    class="org.databene.platform.fixedwidth.FixedWidthEntitySource">
  <property name="uri" value="shop/products.import.fcw"/>
  <property name="entity" value="product"/>
  <property name="properties" value="ean_code[13],name[30],price[8r0]"/>
</bean>

<iterate type="product" source="products_file" consumer="ConsoleExporter"/>
```

5.23 Consumers

Consumers are the objects that finally receive the data after creation, conversion and validation. Consumers can be files, storage systems or custom JavaBeans that implement the Consumer interface. They are not supposed to mutate generated data. That is reserved to Converters.

5.23.1 Specifying Consumers

A <generate> element may have consumers in <consumer> sub-elements or in a comma-separated list in a consumer attribute, e.g. consumer="a,b". A consumer sub element has the same syntax as a <bean> element, e.g.

```
<generate type="db_product">
  <consumer class="com.my.SpecialConsumer">
    <property name="format" value="uppercase"/>
```

```
  </consumer>
</generate>
```

A consumer attribute may hold a comma-separated list consisting of

- names of previously defined beans
- fully qualified class names of consumer implementations

Examples:

```
<database id="db" .../>
```

The <database> declaration will be described later.

```
<bean id="special" class="com.my.SpecialConsumer"/>
  <property name="format" value="uppercase"/>
</bean>

<generate type="db_product" consumer="db,special"/>
```

or

```
<bean id="special" class="com.my.SpecialConsumer"/>
  <property name="format" value="uppercase"/>
</bean>


<generate type="db_product" consumer="com.my.SpecialConsumer"/>
```

5.23.2 Consumer Life Cycle

In most cases Consumers relate to heavyweight system resources, so it is important to know their life cycle. There are two different types of life cycle:

- Global Consumer: A consumer defined as a <bean> has a global scope (thus is called 'global consumer') and is closed when Benerator finishes. So you can use the same consumer instance for consuming output of several different <generate> and <iterate> blocks.
- Local Consumer: A consumer defined 'on the fly' in a generate/iterate block (by 'new', class name or <consumer>) has a local scope and is immediately closed when its generate/iterate block finishes. If you want to generate a file and iterate through it afterwards you need to have it closed before. The most simple way to assure this is to use a local consumer in file generation.

5.24 Exporting Data to Files

You will need to reuse some of the generated data for setting up (load) test clients. You can export data by simply defining an appropriate consumer:

```
<import platforms="fixedwidth" />
<generate type="db_product" consumer="db">
  <consumer class="FixedWidthEntityExporter">
    <property name="uri" value="products.fcw"/>
    <property name="properties" value="ean_code[13],name[30l],price[10r0]"/>
```

```
  </consumer>
</generate>
```

5.25 Post Processing Imported or Variable Data

When importing data or using helper variables, you may need to overwrite imported attributes. You can do so by

- **overwriting** them (e.g. with a generated ID value) or
- manipulating the imported attribute value with a **script** (e.g. replacing CARD=Y/N with 0/1) or
- using a **map** to convert between predefined values or
- using a **converter** to transform attributes individually (e.g. for converting strings to uppercase)

You could also combine the approaches

5.25.1 overwriting post processing

```
<iterate type="TX" source="tx.ent.csv" >
  <id name="ID" generator="IncrementalIdGenerator" />
</generate>
```

5.25.2 "script" post processing

```
<iterate type="TX" source="tx.ent.csv">
  <attribute name="CARD" script="TX.CARD == 'Y' ? 1 : 0" />
</generate>
```

5.25.3 "map" post processing

For mapping imported (or generated) values, you can use a convenient literal syntax, listing mappings in a comma-separated list of assignments in the form original_value -> mapped_value. Values need to be literals here too, so don't forget the quotes around strings and characters! This is a postprocessing step, so it can be combined with an arbitrary generation strategy.

Example

```
<iterate type="db_user" source="db">
    <variable name="p" generator="person" />
    <attribute name="gender"
        script="p.gender.name()"
        map="'MALE'->'m','FEMALE'->'f'" />
</iterate>
```

In a script, the keyword **this** refers to the entity currently being generated/iterated.

5.25.4 "converter" post processing

For more intelligent/dynamic conversions, you can inject a converter, e.g. for converting strings to upper case:

```
<iterate type="TX" source="tx.ent.csv">
    <attribute name="PRODUCT"
        script="{this.PRODUCT}"
        converter="CaseConverter" />
```

```
</generate>
```

5.26 Anonymizing Production Data

When importing data from production systems, you can anonymize it by overwriting its attributes as described in 'post processing import data'. If you need to assure multi-field-dependencies when overwriting, you can choose a prototype-base approach: import data from one source and merge it with prototypes that are generated or imported from another source.

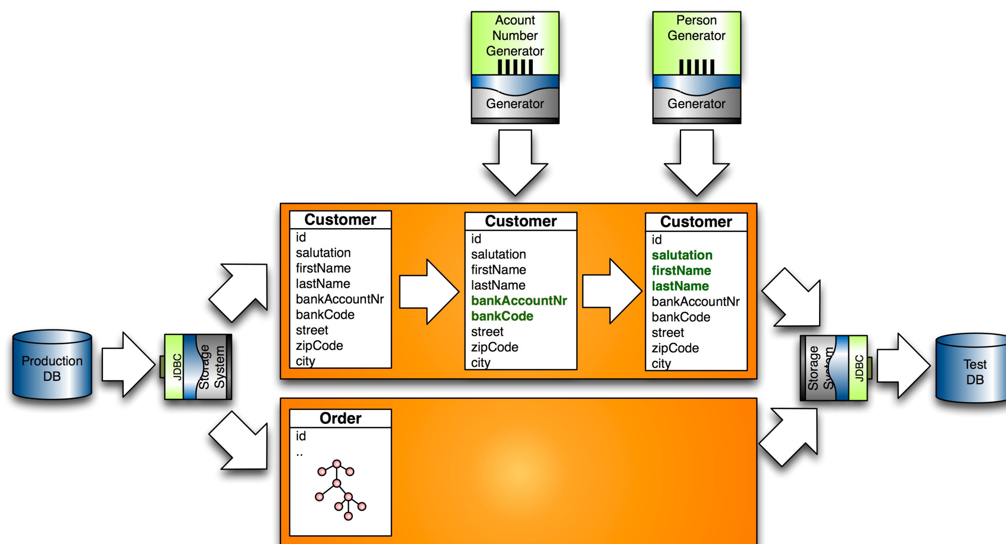
In the following example, customers are imported from a database table in a production database (prod_db), anonymized and exported to a test database (test_db). All attributes that are not overwritten, will be exported as is. Since customer names and birth dates need to be anonymized, a prototype generator (...PersonGenerator) is used to generate prototypes (named person) whose attributes are used to overwrite production customer attributes:

```
<iterate source="prod_db" type="db_customer" consumer="test_db">

    <variable name="person"
        generator="org.databene.domain.person.PersonGenerator"/>

    <attribute name="salutation" script="person.salutation" />
    <attribute name="first_name" script="person.givenName" />
    <attribute name="last_name"  script="person.familyName" />
    <attribute name="birth_date" nullable="false" />

</iterate>
```



5.27 "condition"

When anoymizing or importing, data one sometimes meets difficult multi-field-constraints of the form „if field A is set then field B must be set and field C must be null“. It many cases, an easy solution is to import data, mutate only non-null fields and leave null-valued fields as they are. A short syntax element to do so is the condition attribute. It contains a condition and when added to a component generator, the generator is only applied if the condition resolves to true:

```
<iterate source="db1" type="customer" consumer="">

    <attribute name="vat_no" condition="this.vat_no != null"
           pattern="DE[1-9][0-9]{8}" unique="true" />

</iterate>
```

5.28 Converters

Converters are useful for supporting using custom data types (e.g. a three-part phone number) and common conversions (e.g. formatting a date a date as string). Converters can be applied to entities as well as attributes by specifying a converter attribute:

```
<generate type="TRANSACTION" consumer="db">
    <id name="ID" type="long" strategy="increment" param="1000" />
    <attribute name="PRODUCT" source="{TRANSACTION.PRODUCT}"
        converter="CaseConverter"/>
</generate>
```

For specifying Converters, you can

- use the class name
- refer a JavaBean in the benerator context
- provide a comma-separated Converter list in the two types above

benerator supports two types of converters:

- Classes that implement the interface org.databene.commons.Converter
- Classes that extend the class java.text.Format

If the class has a 'pattern' property, benerator maps a descriptor's pattern attribute to the bean instance property.

5.29 Validators

Validators assist you in assuring validity of generated data. Validators can be applied to attributes and full entities. They intercept in data generation: If a generated item is invalid, it will be discarded and regenerated transparently. This is a cheap way of fulfilling complex constraints which are only partially known: If you have a class or system that can validate this data, you can set up a heuristic generation which has a high probability of succeeding and simply discard the invalid ones. If the ratio of invalid objects is more than 99%, benerator will give you a warning since this is likely to impact generation performance. If the ratio rises to 99.9%, benerator will terminate with an exception.

For specifying Validators, you can

- use the class name
- refer a JavaBean in the benerator context
- provide a comma-separated Validator list in the two types above

5.30 Creating random Entities

Entities can be generated without any input files - benerator provides a rich set of Generator implementations. When using <generate> without a 'source' attribute, the registered systems (e.g. the database are requested for metadata). From the metadata, attributes are generated that match the metadata (e.g. database) constraints, as column length, referenced entities and more. By default, associations are treated as one-to-one associations.

With benerator's many useful defaults, you have a minimum effort on initial configuration:

```
<generate type="db_user" count="1000" consumer="db" />
```

Id generation defaults to an increment strategy and for all other columns useful defaults are chosen.

Entities are generated as long as each attribute generator is available and limited by the number specified in the 'count' attribute. The 'pageSize' defines the number of creations after which a flush() is applied to all consumers (for a database system this is mapped to a commit).

5.31 Entity Count

There are different ways of determining or limiting the number of generated entities:

- the count attribute specifies a fix number of instances to create
- the minCount, maxCount and countDistribution attributes let benerator choose an instance count with the specified characteristics.
- availability of the component generators

Data generation stops if either the limit count is reached or a component generator becomes unavailable.

If you have problems with unexpectedly low numbers of generated entities you can set the log category org.databene.benerator.STATE to debug level as described in ???.

5.32 Using Predefined Entities

When iterating predefined entities (e.g. imported from file or database), benerator's default behaviour is to serve each item exactly once and in the order as provided. You can change that behaviour in many ways, but need to be aware of the iterated data volume:

For small data sets (< 100,000 items) you can apply a distribution method (see Section 3.29, “Applying a Weight Function” or Section 3.30, “Applying a Sequence”). This will cause benerator to load all available instances into memory and serve them as specified by the distribution: A WeightFunction will tell benerator

how often to serve an instance of a certain list index, a Sequence will tell each index consecutively. Depending on the Sequence, data can be provided uniquely or weighted.

For big data sets (> 100,000 items) you need to be more conservative, since the data volume is not supposed to fit into main memory. You have two options here: cyclic iteration and proxy iteration. Actually, both types can be combined.

5.33 Iterating Predefined Entities Consecutively

By default, imported entities are processed consecutively and only once.

When setting cyclic="true" benerator serves the imported data consecutively too but does not stop when it reaches the end. Instead it restarts iteration. Beware: For SQL queries this means that the query is reissued, so it may have a different result set than the former invocation.

When using a distribution, you can manipulate what happens with the original data, e.g. by dropping or repeating data.

5.34 Applying a Weight Function

You can weigh any arbitrary imported or numeric data by a Weight Function. A Weight Function is defined by a class that implements the interface org.databene.model.function.WeightFunction:

```
public interface WeightFunction extends Weight {
    double value(double param);
}
```

When using a weight function, benerator will serve data items in random order and as often as implied by the function value. benerator automatically evaluates the full applicable number range (as defined by numerical min/max or number of objects to choose from) and normalize the weights. There is no need to provide a pre-normalized distribution function. You may define custom Weight Functions by implementing the WeightFunction interface.

5.35 Applying a Sequence

A Sequence is basically a number generator. It can provide a custom random algorithm, a custom weighted number generator or a unique number generation algorithm.

For a list of predefined sequences, see *Section 8.3, “Sequences”*. The definition of a custom sequence is described in *Section 9.5, “Custom Sequences”*.

5.36 Importing Weights

When importing data from data sources, you have additional options for specifying weights. They are different when importing simple data or entities.

5.36.1 Importing primitive data weights

When importing primitive data from a CSV file, each value is expected to be in an extra row. If a row has more than one column, the content of the second column is interpreted as weight. If there is no such column, a weight of 1 is assumed. benerator automatically normalizes over all data objects, so there is no need to care about manual weight normalization. Remember to use a filename that indicates the weight character, using a suffix like '.wgt.csv' or 'wgt.xls'.

If you, for example, create a CSV file roles.wgt.csv:

```
customer,7
```

```
clerk,2
admin,1
```

and use it in an configuration like this:

```
<generate type="user" count="100">
    <attribute name="role" source="roles.wgt.csv" />
</generate>
```

this will create 100 users of which about 70 will have the role 'customer', 20 'clerk' and 10 'admin'.

5.36.2 Weighing imported entities by attribute

when importing entities, an entity attribute can be chosen to represent the weight by specifying distribution="weighted[attribute-name]". Remember to indicate, that the source file contains entity data by using the correct file suffix, e.g. '.ent.csv' or '.ent.xls'

Example: If you are importing cities and want to weigh them by their population, you can define a CSV file cities.ent.csv:

```
name,population
New York,8274527
Los Angeles,3834340
San Francisco,764976
```

and e.g. create addresses with city names weighted by population, when specifying

```
  <generate type="address" count="100" consumer="ConsoleExporter">
    <variable name="city_data" source="cities.ent.csv"
        distribution="weighted[population]"/>
    <id name="id" type="long" />
    <attribute name="city" script="city_data.name"/>
  </generate>
```

5.37 Nesting Entities

Entities can form composition structures, which are generated most easily by recursive <generate> structures.

Consider a database schema with a db_user and a db_customer table. Each row in the db_customer table is supposed to have a row with the same primary key in the db_user table. So an easy way to implement this is to nest db_customer generation with db_user generation and use the outer db_user's id value for setting the db_customer id:

```
<generate type="db_user" count="10" consumer="db">
      <id name="id" strategy="increment" />
      ...
      <generate type="db_customer" count="1" consumer="db">
            <attribute name="id" script="{db_user.id}" />
            ...
      </generate>
</generate>
```

5.38 Imposing one-field business constraints

Simple constraints, e.g. formats can be assured by defining an appropriate Generator or regular expression, e.g.

```
<import domains="product" />
<!-- create products of random attribs & category -->
<generate type="db_product" count="1000" pageSize="100">
    <attribute name="ean_code" generator="EANGenerator"/>
    <attribute name="name" pattern="[A-Z][A-Z]{5,12}"/>
    <consumer ref="db"/>
</generate>
```

5.39 Imposing multi-field-constraints

For supporting multi-field-constraints, you can use a prototype-based approach: Provide a Generator by a variable element. This generator creates prototype objects (or object graphs) which are used as prototype. They may be Entities, JavaBeans or Maps. For example, this may be an importing generator. On each generation run, an instance is generated and made available to the other sub generators. They can use the entity or sub elements by a source path attribute:

```
<import domains="person"/>
<generate type="db_customer" consumer="db">
    <variable name="person" generator="PersonGenerator"/>
    <attribute name="salutation" script="person.salutation"/>
    <attribute name="first_name" script="person.givenName"/>
    <attribute name="last_name" script="person.familyName"/>
</generate>
```

The source path may be composed of property names, map keys and entity features, separated by a dot.

5.40 Default Attribute Settings

Usually most entities have common attribute names, e.g. for ids or audit data. You can specify default settings by column name:

```
<defaultComponents>
    <id name="ID" type="long" source="db"
        strategy="sequence" param="hibernate_sequence"/>
    <attribute name="VERSION" values="1"/>
    <attribute name="CREATED_AT" generator=currentDateGenerator"/>
    <attribute name="CREATED_BY" values="vbergmann"/>
    <attribute name="UPDATED_AT" generator="currentDateGenerator"/>
    <attribute name="UPDATED_BY" values="vbergmann"/>
</defaultComponents>
```

If a table has a column which is not configured in the benerator descriptor but as defaultComponent, benerator uses the defaultComponent config. If no defaultComponent config exists, benerator falls back to a useful standard setting.

5.41 Settings

You can define global settings in the descriptor file:

```
<setting name="my_name" value="Volker" />
```

or import several of them from a properties file:

```
<include uri="my.properties" />
```

5.42 Querying Information from a System

Arbitrary information may be queried from a system by a 'selector' attribute, which is system-dependent. For a database SQL is used:

```
<generate type="db_order" count="30" pageSize="100">
    <reference name="customer_id" source="db"
        selector="select id from db_customer" cyclic="true"/>
    <consumer ref="db"/> <!-- automatically chosen by benerator -->
</generate>
```

Using cyclic="true", the result set will be re-iterated from the beginning when it has reached the end.

You may apply a distribution as well:

```
<generate type="db_order_item" count="100" pageSize="100">
   <attribute name="number_of_items" min="1" max="27" distribution="cumulated"/>
   <reference name="order_id" source="db"
      selector="select id from db_order" cyclic="true"/>
   <reference name="product_id" source="db"
      selector="select ean_code from db_product" distribution="random"/>
   <consumer ref="db"/>
</generate>
```

The result set of a selector might be quite large, so take care, which distribution to apply:

When using weights, the complete result set is loaded into RAM. Such a distribution should not be applied to result sets of more than 100.000 elements (this applies for most sequences as well). A WeightFunction should be restricted to at most 10.000 elements.

'Unlimited' sequences are

- 'expand'
- 'randomWalk'
- 'repeat'
- 'step'

You can use script expressions in your selectors, e.g.

```
selector="{ftl:select ean_code from db_product where country='${country}'}"
```

The script is resolved immediately before the first generation and then reused. If you need dynamic queries, that are re-evaluated, you can specify them with double brackets:

```
selector="{{ftl:select ean_code from db_product where country='$
{shop.country}'}}"
```

Example:

```
<generate type="shop" count="10">
   <attribute name="country" values="DE,AT,CH"/>
   <generate type="product" count="100" consumer="db">
      <attribute name="ean_code" source="db"
         selector="{{ftl:select ean_code from db_product where country='$
{shop.country}'}}"/>
   </generate>
</generate>
```

5.43 Attribute Metadata Reference

5.43.1 Descriptive Attribute Metadata

name	description	default
name	name of the feature to generate	
type	type of the feature to generate	string
nullable	tells if the feature may be null	true
mode	controls the processing mode: (normal|ignored|secret)	normal
pattern	uses a regular expression for String creation or date format pattern for parsing Dates.	
values	provides a comma-separated list of values to choose from	
unique	whether to assure uniqueness, e.g. unique="true". Since this needs to keep every instance in memory, use is restricted to 100.000 elements. For larger numbers you should use Sequence-based algorithms.	false
min	the minimum Number or Date to generate	1
max	the maximum Number or Date to generate	9
granularity	the resolution of Numbers or Dates to generate	1
minLength	the minimum length of the Strings that are generated	
maxLength	the maximum length of the Strings that are generated	

5.43.2 Constructive Attribute Metadata

name	description	default
generator	uses a Generator instance for data creation	

nullQuota	the quota of null values to create	0
converter	the class name of a Converter to apply to the generated objects	
dataset	a (nestable) set to create data for, e.g. dataset="US" for the United States	
locale	a locale to create data for, e.g. locale="de"	default locale
source	A system, EntityIterator or file to import data from.	
selector	A system-dependent selector to query for data.	
trueQuota	the quota of true values created by a Boolean Generator.	0.5
distribution	the distribution to use for Number or Date generation. This may be a Sequence name or a WeightFunction class name.	
cyclic	auto-resets the generator after it has gone unavailable	false

5.44 Scripts

Scripts are supported in

- descriptor files
- properties files
- DbUnit XML files
- CSV files
- Fixed column width files

A script is denoted by curly braces, e.g. {'Hi, I am ' + my_name}. This syntax will use the default script engine for rendering the text as, e.g. 'Hi, I am Volker'. The default script engine is set writing <setup defaultScript="..."> in the decriptor file's root element. If you want to use different script engines at the same time, you can differ them by prepending the scripting engine id, e.g. {ftl:Hi, I am ${my_name}} or {ben:'Hi, I am ' + my_name}. Scripts in benerator descriptors are evaluated while parsing.

If you need to dynamically calculate data at runtime, use a script attribute, e.g.:

```
<attribute name="message" script="'Hi, ' + user_name + '!'" />
```

In the 'script' attribute, curly braces are not necessary.

Using scripts you can access

- environment variables, e.g. JAVA_HOME
- JVM parameters, e.g. benerator.validate
- any JavaBean globally declared in the benerator setup, e.g. db
- the last generated entity of each type, e.g. db_user
- the entity currently being generated and its attributes, e.g. this.id
- entities generated in outer <generate> elements
- helper variables in the <generate> element, e.g. person.familyName
- predefined or custom FreeMarker methods (when using FreeMarker as script language)
- Static Java methods and attributes, e.g. System.getProperty('user.home')

- instance methods and attributes on objects in the context, e.g. db.system

Variable names used in scripting may not contain points - a point always implies resolution of a local feature of an object, e.g. person.familyName resolves the familyName attribute/property/key of a person.

5.44.1 this

In a script, the keyword 'this' always refers to the entity currently being generated. You can use this to construct attributes which have dependencies to each other:

```
<generate type="product">
    <id name="id" />
    <attribute name="code" script="'ID#' + this.id" />
</generate>
```

5.45 Handling Errors

5.45.1 onError

Several descriptor elements support an onError attribute. It determines an error severity and how Benerator should behave in case of errors.

The default severity is 'fatal', which causes Benerator to stop execution.

Other available severities are ignore, trace, debug, info, warn, error, which mainly influence the log level in which errors are reported, but do not stop execution.

```
<generate type="product" count="1000" onError="fatal" consumer="db">
    <!-- component setup here -->
</generate>
```

5.45.2 BadDataConsumer

For errors that are raised by a consumer, you have the alternative option to catch them and write the data which has caused the error to an alternative consumer. For example, you can write the problematic data to a CSV file named 'errordata.csv' and postprocess it:

```
<generate type="product" count="1000"
  consumer="new BadDataConsumer(new CSVExporter('errors.csv'), db.inserter())">
  <!-- component setup here -->
</generate>
```

Note that this cannot work properly with a database which uses batch processing (see Using Databases).

6 Regular Expression Support

Benerator supports most common regular expression features and given a regular expression, it is able to generate strings that match the regular expression – even in a unique manner if required. Here is a full description of regular expression features recognized for data generation setup:

6.1 Characters

Digits (0-9) and US-ASCII letters (A-Z, a-z) are supported as they are. Special characters can be defined by their UNICODE number in octal or hexadecimal form:

\0n The character with octal value 0n (0 <= n <= 7)

\0nn The character with octal value 0nn (0 <= n <= 7)

\0mnn The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)

\xhh The character with hexadecimal value 0xhh

\uhhhh The character with hexadecimal value 0xhhhh

For control codes in general there is a special notation

\cx The control character corresponding to x (e.g. \cA for Ctrl-A)

and certain control codes have own representations:

\t The tab character ('\u0009')

\n The newline (line feed) character ('\u000A')

\r The carriage-return character ('\u000D')

\f The form-feed character ('\u000C')

\a The alert (bell) character ('\u0007')

\e The escape character ('\u001B')

Some characters have a special meaning in regular expressions, so if you want to use them as a plain character (and not in their regex-meaning), you need to escape them with a backslash:

\. Dot

\- Minus sign

\^ Circumflex

\$ Dollar

\| 'Or' sign

\(Left parenthesis

\) Right parenthesis

\[Left bracket

\] Right bracket

\{ Left curly brace

\} Right curly brace

\\ Backslash character

6.2 Character Classes

A 'character class' defines a set of characters from which one can be chosen and is marked by surrounding brackets: []

[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range). Note: This includes only US-ASCII letters, not special letters of your configured language
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

There are some popular predefined character classes:

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Quantifiers can be used to specify how many characters of a class (or other regular expression construct) should appear:

X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

6.3 Operators

XY	*X* followed by *Y*
X|*Y*	Either *X* or *Y*
(*X*)	X, as a group

6.4 Frequently asked Questions

For generating characters which appear in your language, but not in English (like German umlauts), you can use their unicode representation (e.g. \u00FC for 'ü').

Different implementations of regular expression parsers exist and many have slight differences. So, if you

take a regular expression that worked on one parser and run it on another one, you may get an error message. Benerator users that do not construct a regular expression by themselves, but simply take on 'from the internet' observe the same effect: The most frequent fault is if someone wants to generate a character that has a special meaning in a regular expression and does not escape it with a backslash, e.g. \., \\, \-, \|, \[, \], \{, \}, …

An example: Some regex parsers recognize that the expression [A-]X could resolve to AX or A-. While others (like Benerator's parser) diagnose a malformed character class (a missing character behind the minus) and report an error. You can resolve this by escaping the minus sign: [A\-]X.

7 Processing and creating CSV Files

7.1 Iterating entity data from a CSV file

You can iterate entity data from a CSV file by assigning the file with the extension '.ent.csv' and specifying the file name as 'source' in an <iterate> statement, e.g. for printing the data to the console:

```
<iterate type="user" source="user.ent.csv" consumer="ConsoleExporter"/>
```

This way, you need to have a CSV file which uses column headers and the default column separator (which is comma by default and can be set globally in the root element's defaultSeparator attribute, e.g. to a semicolon: `<setup defaultSeparator=";">`)

If the CSV file does not have headers or uses another separator or file encoding that deviates from the default, you need to configure the CSV import component (CSVEntitySource) explicitly with a <bean> statement and refer it later:

```
<bean id="in" class="CSVEntitySource">
  <property name="uri" value="headless-in.csv" />
  <property name="separator" value=";" />
  <property name="encoding" value="UTF-8" />
  <property name="columns" value="name,age" />
</bean>

<iterate type="user" source="in" consumer="ConsoleExporter"/>
```

For CSV files without header, you need to specify a comma-separated list of column names in the 'columns' property.

7.2 Creating CSV files

For creating a CSV file you must always take the same approach as above: Defining a bean with its properties and refering it as consumer:

```
<bean id="out" class="CSVEntityExporter">
   <property name="uri" value="target/headless-out.csv" />
   <property name="columns" value="name, age, check" />
</bean>

<generate type="product" count="200" consumer="out">
   …
</generate>
```

See the component documentation of CSVEntityExporter for more details.

8 Using Relational Databases

8.1 Import and <database>

For using database-related features, you must import the 'db' package:

```
<import platforms="db" />
```

You can easily define a database:

```
<database id="db" url="jdbc:hsqldb:hsql://localhost"
  driver="org.hsqldb.jdbcDriver" user="sa" batch="false"/>
```

A database must have an id by which it can be referenced later. For starting a project, it is better to have batch="false". In this mode, database errors are easier to track.

The following attributes are available in the <database> element:

name	description
id	identifier under which the database is accessible in the context (required)
environment	Identifier of a database repository entry
url	JDBC database url
driver	JDBC driver class
user	user name for login
password	password for login
catalog	Database catalog to use
schema	database schema to use
includeTables	Regular expression for tables to be used
excludeTables	Regular expression for tables to be ignored
lazy	boolean flag to enable lazy metadata parsing. This improves performance on large systems of which only a small number of tables is actually used in generation.
metaCache	boolean flag which can be activated on databases with slow database access to cache database metadata on the local file system instead of reparsing it on each run
batch	boolean flag to specify if batch inserts and updates shall be done
fetchSize	JDBC fetch size for query results
readOnly	indicates if only read access shall be allowed in order to protect sensitive data
acceptUnknown ColumnTypes	If set to true, Benerator accepts exotic database column types without complaining and relies on the user to take care of the appropriate data type when generating values for the column.

Attention: Benerator has some built-in knowledge about the most widely used database systems and their conventions. So in most cases, it is sufficient to provide url, driver user and password. In special cases, e.g. if you want to access a schema ehich is not the default schema of your user, you may have to set schema (and possibly catalog) explicitly.

8.2 Usual Database Settings

Database	URL Format	Driver Class	Default Catalog	Default Schema
DB2	jdbc:db2://host:50001/dbname	com.ibm.db2.jcc.DB2Driver		<user name>
Derby	jdbc:derby://host:1527/dbname			<user name>
Firebird	jdbc:firebirdsql:host/3050:dbname	org.firebirdsql.jdbc.FBDriver		
H2	Jdbc:h2:....	org.h2.Driver		PUBLIC
HSQL	jdbc:hsqldb:hsql://host:9001/dbname	org.hsqldb.jdbcDriver		PUBLIC
MySQL	jdbc:mysql://host:3306/dbname	com.mysql.jdbc.Driver	<user name>	
Oracle	jdbc:oracle:thin:@host:1521:SID	oracle.jdbc.driver.OracleDriver		<user name>
Postgres	jdbc:postgresql://host:5432/dbname	org.postgresql.Driver		<user name>
SQL Server	jdbc:jtds:sqlserver://host:1433; DatabaseName=dbname	net.sourceforge.jtds.jdbc.Driver	dbo	

8.3 Using Database Repositories

For frequently-used databases it is more convenient to use a central database configuration repository. The repository is located in a folder 'databene' under your user home directory. You can define a database configuration with a name (e.g. 'mydb') by storing a correspondingly named properties file there assigning the suffix '.env.properties' (e.g. 'mydb.env.properties', on Windows the file location would be C:\Documents and Settings\<user_name>\mydb.env.properties). In the file you can configure the JDBC connection information with the keys db_url, db_driver, db_user, db_password and db_url.

As an example, a file mydb.env.properties would configure the environment 'mydb' and would have a content like this for an HSQL database:

```
db_url=jdbc:hsqldb:mem:DbRelatedTest
db_driver=org.hsqldb.jdbcDriver
db_user=sa
db_password=
db_schema=public
```

Having done so, you can connect a database more simply using the <database>'s 'environment' attribute:

```
<database id="db" environment="mydb"/>
```

If you define a mydb.env.properties file in the directory in which Benerator executes, this file will be used. If not, the configuration is taken from your database repository.

If you add conflicting attributes in your <database> element (like another user and password), they override the configuration details in the database repository. This way you can have a central and convenient database lookup and can access the database with different users in the same run. An example:

```
<database id="db1" environment="mydb" user="user1" password="password1"/>

<database id="db2" environment="mydb" user="user2" password="password2"/>
```

8.4 Caching Database Metadata

On very large databases, especially when accessed remotely, database metadata retrieval may take several minutes. In this case, you can make use of the metaCache facility.

Two preconditions exist for using meta data caching: You need to

1. configure a database repository (environment) and
2. set metaCache to "true"

```
<database id="db2" environment="mydb" metaCache="true"/>
```

On the first run, you will not observe any speedup – meta data parsing may even take longer, since the cache needs to be built up on the first run and Benerator is likely to read (much) more meta data than you absolutely need for your specific data generation. When done so, Benerator saves the meta data in an XML file. On subsequent runs, Benerator notices the cache file and reads it within milliseconds.

Cache Invalidation: There are several reasons that make Benerator invalidate its cache information and reload it:

- **Execution of SQL code via <execute>**: Benerator is not so clever about interpreting SQL code. Thus it interprets any executed SQL code as a potential meta data change and invalidates the cache.
- **Cache time out**: If the cache file is older than twelve hours, Benerator throws it away just in case. If you are sure that the database has not changed meanwhile, you can perform a 'touch' on the cache file.

Warning: If you change the database structure from another client system and Benerator is configured to cache meta data, there is no way to become aware of it and the old meta data cache file is used which has become obsolete. You need to delete the cache file manually in such cases!

8.5 Executing SQL statements

SQL code can be executed, e.g. from a file:

```
<execute uri="drop-tables.sql" target="db" onError="warn"/>
```

Uris are resolved relative to the benerator file that declares them (as common in HTML). If the file is not found locally, it is searched relative to the current working directory.

You can inline SQL code as well:

```
<execute target="db" onError="warn">
  CREATE TABLE db_role (
    id   int         NOT NULL,
    name varchar(16) NOT NULL,
    PRIMARY KEY (id)
  );
</execute>
```

8.5.1 Alternative Delimiters

By default, the semicolon is the delimiter between commands: Benerator splits SQL commands by their delimiter and sends one after the other to the database. In some cases you need a different behaviour, e.g. if a procedure should be defined and/or called. In such cases, you can specify an alternative delimiter in an <execute> statement:

```
<execute target="db" separator="/">
    declare
```

```
        output_var varchar(500);
    begin
        EXECUTE_PROC_A(output_var);
        EXECUTE_PROC_B(output_var);
    end;
    /
</execute>
```

8.6 Inserting entities into a database

When using a database as consumer in a <generate> or <iterate> element, the elements are *inserted* by default. For information, how to *update* entries, see the next chapter.

```
<generate type="db_user" count="50000" consumer="db" />
```

If primary key generation should be performed by the database, you need to tell benerator to ignore the field, setting the mode to 'ignored'

```
<generate type="db_user" count="50000" consumer="db" >

    <id mode="ignored" />

</generate>
```

8.7 Database-related Id Generators

The following id generators make use of database features:

- **DBSequenceGenerator**: Retrieves id values from a database sequence. With default settings, it operates quite slowly, since it incurs an additional database call for obtaining the id value for each generated entity. When setting its property '**cached**' to true, it fetches the current value of its database sequence, creates ids offline in Benerator RAM and updates the database sequence in the end. Of course this requires Benerator to run in a single instance and no other client may be writing data to the system while Benerator is generating – otherwise a primary key conflict may arise.
- **DBSeqHiLoGenerator**: Combines a value retrieved from a database with a local counter to create unique values (with a strongly reduced performance burden compared to the DBSequenceGenerator)
- **QueryGenerator**: Uses a database query to calculate id values
- **QueryLongGenerator**: Uses a database to calculate id values of number type
- **QueryHiLoGenerator**: Works like a DBSeqHiLoGenerator, but based on a query instead of a sequence
- **SequenceTableGenerator**: Lets you read and increment values from database tables

Best performance with cluster-safe generators is achieved with the DBSeqHiLoGenerator, followed by the QueryHiLoGenerator.

8.7.1 SequenceTableGenerator

The SequenceTableGenerator lets you create unique long values from a database table. Depending on the table structure this can be trivial or tricky.

You always need to specify a **database**, a **table** and a **column** from which to read the value, in non-trivial cases, you also need a **selector**.

Single-Value-Table

In the simplest case, you have a table which stores a single row with a single value:

```
<database id="db" environment="mydb" />

<bean id="sg" class="SequenceTableGenerator">
   <property name="database" ref="db">
   <property name="table" value="MY_TABLE">
   <property name="column" value="SEQ_VALUE">
</bean>

<generate type="PERSON" count="100" consumer="db">
   <id name="id" type="long" generator="sg" />
   ...
</generate>
```

Name-Value-Pair Table

In a slightly more difficult case, you have name-value-pairs of 'sequence identifier' and 'sequence value'. Then you must specify a selector, that tells Benerator which row to use. For example, if the sequence for the PERSON table is specified by a row in which the SEQ_ID column has the value 'PERSON':

```
<database id="db" environment="mydb" />

<bean id="sg" class="SequenceTableGenerator">
   <property name="database" ref="db">
   <property name="table" value="MY_TABLE">
   <property name="column" value="'SEQ_VALUE">
   <property name="selector" value="SEQ_ID = 'PERSON'">
</bean>

<generate type="PERSON" count="100" consumer="cons">
   <id name="id" type="int" generator="sg" />
</generate>
```

Arbitrary Table

You can support arbitrary complex sequence tables with a **parameterized selector**. It marks each parameter with a question mark (**?**) and must be invoked differently than the examples above, using a **script** that calls the **generateWithParams(...)** method:

```
<database id="db" environment="mydb" />

<bean id="sg" class="SequenceTableGenerator">
   <property name="database" ref="db">
   <property name="table" value="MY_TABLE">
   <property name="column" value="'SEQ_VALUE">
   <property name="selector" value="MOD_ID = ? and ITEM_ID = ?">
</bean>

<generate type="PERSON" count="100" consumer="cons">
   <!-- calculate mid and iid as required -->
   <id name="id" type="int" script="sg.generateWithParams(mid, iid)" />
</generate>
```

8.8 Handling of common Columns

In many databases, you encounter common columns like auditing information 'created_by', 'created_at', 'updated_by', 'updated_at' or optimistic locking columns. See the chapter 'Default Attribute Settings' for instructions how to define a common default generation settings for these.

8.9 Determining attribute values by a database query

You can use database queries to determine column values. A **source** attribute identifies the database to query and a **selector** the SQL query to perform:

```
<attribute name="user_rank"
    source="db"
    selector="select rank from ranks where active = 1" />
```

You can use **source** and **selector** in <attribute>, <id>, <reference> and <variable> statements.

Attention: Whis this syntax, the query's result set is iterated throughout the <generate> loop until its end is reached. In the example above, a result set with the rows [1], [2], [3] will result in the user_rank values 1 for the first generated entry, 2, for the second and 3 for the third. After that the end of the result set is reached, the component signals, that it is unavailable and Benerator will terminate the generation loop. If you configured more than 3 objects to be generated, you will get an exception that Benerator was not able to provide the requested number of data sets. You have the following alternatives:

1. Cycling through the result set again and again (cyclic="true")
2. Apply a distribution choose a mode for selecting elements of the result set repeatedly or randomly
3. If the query should be performed for each generated entity and is supposed to provide a single result, this is called sub query and is supported by a 'subQuery' attribute

8.9.1 Cycling through query result sets

When using a **selector** in combination with **cyclic="true"**, the query is automatically repeated when the end of the result set is reached:

```
<attribute name="user_rank"
    source="db"
    selector="select rank from ranks where active = 1"
    cyclic="true" />
```

8.9.2 Applying a distribution to a query

When using a **selector** in combination with a **distribution**, the query's result set in processed by the selected distribution algorithm. Depending on the distribution, the result set may be buffered. Result set elements may be provided uniquely or repeatedly in an ordered or a random fashion. See the distribution reference. The most common distribution is the 'random' distribution which buffers the full result set and then provides a rnadomly chosen entry on each invocation.

```
<attribute name="user_rank"
    source="db"
    selector="select rank from ranks where active = 1"
    distribution="random" />
```

8.9.3 Sub selectors

Frequently you will encounter multi-field-constraints in an entity which can be matched by a query. Usually this means to first generate a random value and then to impose a database query with the value generated before. The query's results are valid only for the currently generated entity, and in general only one query result row is expected. You have a kind of „sub query" which is handled best by using a **subSelector**. For example, you might offer products (table 'product') in different geographical regions and have a cross-reference table product_region that describes, which products are available in which region:

```
<attribute name="region"
    values="'americas','emea','asia'" />

<reference name="product"
    source="db"
    subSelector="{{'select product_id from product_region
                  where region_id = ' + this.region }}"
/>
```

Attention: In the example, you need double brackets {{…}} in order to signal that **this.region** should be reevaluated on each invocation. When using a single bracket, the query memorizes the value of **this.region** at the first invocation and reuses it on each subsequent call.

8.10 Resolving Database Relations

8.10.1 Automatic referencing

By default, benerator assumes that all relations are one-to-one as the most defensive choice. Thus the following setup in which a table db_user references a table db_role will cause an error:

```
<generate type="db_role" count="10" consumer="db" />

<generate type="db_user" count="100" consumer="db" />
```

This is because, assuming a one-to-one relationship, you can only generate as many users as unique roles are available! ...and you have generated only 10 roles before. In other words, in fully automatic data generation, the number of user entries will be the number of role entries.

In most cases you actually deal with many-to-one relationships and thus need to specify its characteristics explicitly, typically by using a distribution. Basically, a reference is defined by (column) **name**, (dabase) **source** and **targetType** (referenced table):

```
<generate type="db_role" count="10" consumer="db" />

<generate type="db_user" count="100" consumer="db">

    <reference name="role_fk" targetType="db_role" source="db"
       distribution="random"/>

</generate>
```

This will cause creation of 100 users which are evenly distributed over the roles.

8.10.2 Null references

If you want to generate only null values, you can reduce the declaration to a **name** and **nullQuota="1"** element:

```
<generate type="db_user" count="100" consumer="db">

    <reference name="role_fk" nullQuota="1"/>

</generate>
```

8.10.3 Selective Referencing

For restricting the objects referred to or using an individual way to construct the reference values, you can specify a selector which will be evaluated by the target system and return a reference value. For databases the selector needs to be a SQL where clause or complete query:

```
<generate type="db_role" count="10" consumer="db" />

<generate type="db_user" count="100" consumer="db">

    <reference name="role_fk" targetType="db_role" source="db"
        selector="role_name != 'admin'" distribution="random"/>

</generate>
```

8.10.4 Other referencing options

Besides selective referencing, you can use (almost) the full feature set of <attribute> elements to generate references, e.g. constant, pattern, values, script, etc. You could, e.g., configure the use of each role type by itself:

```
<generate type="db_user" count="5" consumer="db">
    <reference name="role_fk" constant="'admin'"/>
    ...
</generate>

<generate type="db_user" count="95" consumer="db">
    <reference name="role_fk" constant="'customer'"/>
    ...
</generate>
```

8.11 Composite Keys

Benerator does not provide an **automated** composite key handling, but you can configure it to handle them **explicitly**. The typical approach for this is a prototype query.

8.12 Prototype Queries

For the general idea of prototype-based generation, see the corresponding chapter. In addition to the core features, the prototype approach is a good way to handle composite primary keys and composite foreign keys, since their components are available in combination.

8.12.1 Prototype Queries on Entities

When querying entities, you specify the database to query as **source**, the where clause of the select as **selector**, and the table which is queried as **type**. After that, you can access the results' attributes by their names:

```
<variable name="_product" type="MY_PRODUCT"
      source="db" selector="sysdate between VALID_FROM and VALID_TO"
      distribution="random" />
<reference name="PRODUCT_ID" script="_product.PRODUCT_ID" />
<attribute name="ROUTING_TYPE" script="_product.ROUTING_TYPE" />
```

When aggregating data with a general query that is not (or cannot be) mapped to a type, you can access the results' column values as array elements (indices are 0-based):

```
<variable name="_product" source="db"
      selector="select PRODUCT_ID, ROUTING_TYPE from MY_PRODUCT
            where sysdate between VALID_FROM and VALID_TO"
      distribution="random" />
<reference name="PRODUCT_ID" script="_product[0]" />
<reference name="ROUTNG_TYPE" script="_product[1]" />
```

8.13 Exporting Database Content

The rows of a database table can be iterated simply. Here's an example for writing all users of a table 'db_user' to a file 'users.csv':

```
<iterate source="db" type="db_user"
```

```
      consumer="new CSVEntityConsumer('users.csv')" />
```

You can as well select a subset to iterate:

```
<iterate source="db" type="db_user" selector="active = 1"
      consumer="new CSVEntityConsumer('users.csv')" />
```

8.14 Updating Database Entries

When using a database, you can update existing entries.

For example, if you have db_orders that refer (one-to-many) db_order_items, you can calculate the sum of the db_order_items' total_price values and write it to the db_order's total_price column:

```
<iterate type="db_order" source="db" consumer="db.updater()">

    <attribute name="total_price" source="db"

        selector="{{ftl:select sum(total_price) from db_order_item
            where order_id = ${db_order.id}}}"

        cyclic="true"/>

</iterate>
```

The described update mechanism can also be used to anonymize production data – see the chapter 'Production Data Anonymization'. For transfering user data from a source database 'sourcedb' to a target database 'testdb', you would write

```
<iterate source="sourcedb" type="db_user" consumer="testdb">

    <!-- anonymize here -->

</iterate>
```

If you have duplicated the database and want to anonymize the copy by updating the tables, you would write

```
<iterate type="db_user" source="db" consumer="db.updater()">

    <!-- anonymize here -->

</iterate>
```

If you want to read data from one table, anonymize it and write it to a different table in the same database, you can use a special inserter:

```
<iterate type="prod_user" source="db" consumer="db.inserter('anon_user')">

    <!-- anonymize here -->

</iterate>
```

8.15 Controlling Transactions

By default, Benerator performs one transaction per data set that is generated. When generating huge amounts of data, transaction overhead quickly becomes significant and you will want to insert several data set in a common transaction. You can use the pageSize argument to configure the number of data elements per transaction. For most databases and tasks, pageSize="1000" is a reasonable setting:

```
<generate type="user" count="1000000" pageSize="1000" consumer="db" />
```

For further hints on improving performance, refer to the 'Performance' section.

If you are nesting creation loops, you can set the transaction control for each level separately:

```
<generate type="user" count="1000" pageSize="100" consumer="db">
      <generate type="order" count="50" pageSize="500" consumer="db" />
</generate>
```

But an 'inner' transaction commit will commit the outer elements too, so you may get more transactions than you expect. The inner pageCount in the descriptor example makes the outer pageSize 10 effectively, since there is a commit after 500 orders. With 50 orders per customer, it is a commit for every 10th customer.

In most cases it is feasible and more intuitive to make the sub creation loops simply join the outer transaction control, by setting their pageSize to zero:

```
<generate type="user" count="1000" pageSize="100" consumer="db">
      <generate type="order" count="50" pageSize="0" consumer="db" />
</generate>
```

Any <generate> loop with pageSize > 0 is flushed when finished. For databases this means a commit.

8.16 Transcoding Database Data

Benerator's transcoding feature enables you to

1. **copy database entries** from one database to another
2. **assign new primary key values** while copying
3. **transcode relationships** (automatically translate foreign key relationships to the new primary key values)
4. **merge relationships** (make copied data refer to pre-existing data in the target database)

Features 1-3 are can be performed easily, for feature 4 you need so fulfill some preconditions.

8.16.1 Copying database entries

A transcoding task that involves one or more database tables is wrapped with a <transcodingTask> and for each table to be transcoded, there needs to be a <transcode> element that specifies the table name. The <transcode> steps must be specified in the order in which they can be applied without violating foreign key constraints. For example, if you have a table USER which references a table ROLE, you need to transcode ROLE first, then USER.

```
<transcodingTask defaultSource="db1" target="db2">
   <transcode table="ROLE"/>
   <transcode table="USER"/>
</transcodingTask>
```

This copies the full content of the tables ROLE and USER from db1 to db2.

8.16.2 Restricting the copied Set

You can resctrict the set of database entries to copy by using a 'selector' attribute in the <transcode> element:

```
<transcodingTask defaultSource="s" target="t">
   <transcode table="ROLE" selector="ID = 1" />
   <transcode table="USER" selector="ROLE_ID = 1" />
</transcodingTask>
```

Only the ROLE with id 1 and the USERs that refer role #1 are copied.

8.16.3 Transcoding

You can overwrite primary key values and other attributes while you are transfering data using the normal Benerator syntax:

```
<transcodingTask defaultSource="s" target="t">

   <transcode table="ROLE">
      <id name="id" generator="IncrementalIdGenerator" />
   </transcode>

   <transcode table="USER">
      <id name="id" generator="IncrementalIdGenerator" />
   </transcode>

</transcodingTask>
```

Each ROLE and USER gets a new primary key value and the foreign key references from USER to ROLE are reassigned to match the new id values.

8.16.4 Cascaded Transcoding

As an easy approach to transcode graphs of dependent objects along with their parent object, they can be transcoded in cascade along with their 'owner' object. This means, for each transcoded owner object, Benerator looks up, which database rows relate to it as defined in the cascade and transcodes them too. Thus, if you restrict the owners (e.g. company) to a subset of all available owners, the cascade statement assures that only database rows (e.g. department), which relate to this subset (company), are transcoded.

A cascade statement consists of <cascade> element nested in a <transcode> element specifying a ref that tells, which columns of which table make up the table relationship. The Syntax is `table(column1 [, column2 [, …]])`, depending on the number of columns used as foreign key. Benerator looks up the corresponding foreign key constraint in the database and finds out the type of relationship.

As an example, if you want to transcode the rows of a company table and cascade to their departments, you would write

```
   <transcode table="company">
      <cascade ref="department(company_id)" />
   </transcode>
```

Cascades can be nested:

```
   <transcode table="company">
      <cascade ref="department(company_id)">
         <cascade ref="employee(department_id)" />
      </cascade>
   </transcode>
```

In <cascade>, you can overwrite attributes, ids and references like in <transcode>:

```
   <transcode table="company">
      <id name="id" generator="IncrementalIdGenerator" />
      <cascade ref="department(company_id)" />
   </transcode>
```

8.16.5 Merging Foreign Key Relations

Benerator allows you to merge data from different databases. To continue the example above, you could have ROLES and USERS in different databases and merge them into one single target database. This introduces a new requirement: Since you might have automatically assigned technical ids, an object with a given 'business' identity (e.g. user 'Volker Bergmann') might have different 'technical' ids (primary keys of value e.g. 10 or 1000) in different databases. Thus, you need to provide Benerator with a description, which technical id relates to which business id in which database.

There are several alternatives available, so I start with one of the simplest and most widely used in order to give you an overview of the approach and then provide you with a complete list of possibilites.

Let's assume the tables ROLE and USER each have a NAME column with unique identifiers. In this case, you can apply the unique-key identity mapping ans store it in a file with the suffix .id.xml, e.g. identities.id.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbsanity>
   <identity table="ROLE" type="unique-key" columns="NAME" />
   <identity table="USER" type="unique-key" columns="NAME" />
</dbsanity>
```

This identity file enables Benerator to merge foreign key relationships in the transcoding process, for example:

```
<transcodingTask target="db3" identity="identity-def.xml">
   <transcode table="ROLE" source="db1"/>
   <transcode table="USER" source="db2"/>
</transcodingTask>
```

8.16.6 Defining Identities

Identites are definied in a file format taken from DB Sanity. Thus, it is XML and its root element is <dbsanity>. Under the root, all <identity> definitions are listed, each referring to a **table** and having a certain identity **type**, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<dbsanity>
   <identity table="ROLE" type="..." ... />
   <identity table="USER" type="..." ... />
</dbsanity>
```

There are different types of entity definitions available:

natural-pk

The simplest type of identity applies if the primary key (PK) is a business key (natural key). In this case, use a <natural-pk> definition:

```
<identity table="CURRENCY" type="natural-pk" />
```

unique-key

If a non-pk-column or a combination of several columns is unique, use the <unique-key> identity and list the key components in a comma-separated list in a 'columns' attribute:

```
<identity table="PRODUCT" type="unique-key" columns="CATEGORY,CODE"/>
```

You may as well include foreign keys in the list: Benerator will use the business id of the referenced objects to determine the identity of the referer.

nk-pk-query

In more complicated cases, you might need a definition style that gives you more freedom. The <nk-pk-

query> allows you to specify an arbitrary SQL query which is supposed to return a pair of natural key and primary key values for each table row:

```
<identity table="COUNTRY" type="nk-pk-query">
   select
      COUNTRY_NAME.NAME as NK,
      COUNTRY.ID as PK
   from COUNTRY_NAME
   join COUNTRY
      on COUNTRY_NAME.COUNTRY_ID = COUNTRY.ID
   where COUNTRY_NAME.LANGUAGE = 'EN'
</identity>
```

sub-nk-pk-query

A table's rows may have a complex identity definition, which is only unique in the context of a 'parent' row in another table. A good example is if a state name is only unique within a given country, nut different countries may have a state of the same name, but different identities. The provided query must have the parent's primary key as '?' parameter:

```
<identity type="sub-nk-pk-query" table="" parents="COUNTRY">
   select
      sn.STATE_NAME as SUB_NK,
      s.STATE_ID as PK
   from STATE s
   join STATE_NAME sn
      on sn.STATE_ID = s.STATE_ID
   where
      sn.LANGUAGE_ID = 'ENG' AND
      s.COUNTRY_ID = ?
</identity>
```

8.16.7 Limitations

Currently, the amount of data that can be transcoded is limited by the amount of avaliable Java heap memory, but as long as you do not transcode billions of data sets you are not supposed to get problems. Composite primary keys are not supported.

9 Using mongoDB

MongoDB can be used in am manner very similar to relational databases using the mongo4ben plugin (see http://databene.org/mongo4ben). It supports data import from and export of generated data to a mongoDB instance and modification of existing database data. The implementation is still experimental, so do not excpect too much yet. One restriction is for example, that selective queries are not yet supported – all data import refers to the collection by the specified 'type' name and iterates through all elements.

9.1 mongo4ben Installation

Download the mongo4ben distribution from http://databene.org/mongo4ben, extract it and copy the jar files from the distribution's lib folder into your Benerator installation's lib folder.

If you want to distribute a mongoDB data generation project to run on a fresh Benerator installation, put the jars mentioned above into a 'lib' folder inside your project folder.

9.2 Usage

At the beginning, import mongo4ben features in your Benerator project:

```
	<import platforms='mongodb' />
```

Then, a mongodb instance can be declared using a <mongodb> element, using an environment definition file:

```
	<mongodb id='db' environment='mymongo' />
```

The environment definition file has to be in the project folder or at ${user.home}/databene/ and has to carry the name specified in the <mongodb> declaration before, in this case mymongo.env.properties with content like this:

```
	db_url=127.0.0.1:27017
	db_catalog=testDatabase
	db_user=me
	db_password=secret
```

The db_catalog setting has to be set to mongoDB's database name.

That's it! Now you can use mongodb for data import and export.

9.3 Examples

A simple data generation example to help you get started. It generates 100 users with 1 to 3 addresses:

```
<setup>
	<import platforms='mongodb' />
	<mongodb id='db' environment='mymongo' />
	<generate type='mit_user' minCount='100' consumer='db'>
		<attribute name='name' type='string' />
		<attribute name='age' type='int' min='18' max='78' />
		<part name='addresses' container='list' count='2'>
			<attribute name='street' pattern='[A-Z][a-z]{4} Street'/>
			<attribute name='houseNo' type='int' min='2' max='9' />
		</part>
	</generate>
</setup>
```

An example that exports (prints) all user data to the text console:

```
<setup>
	<import platforms='mongodb' />
	<mongodb id='db' environment='mymongo' />
```

```
	<iterate type='mit_user' source='db' consumer='ConsoleExporter'/>
</setup>
```

A trivial anonymization-style example that sets each user's age to 33 and house number to 123:

```
<setup>
	<import platforms='mongodb' />
	<mongodb id='db' environment='mymongo' />
	<iterate type='mit_user' source='db'
			consumer='db.updater(),ConsoleExporter'>
		<attribute name='age' constant='33' />
		<part name='addresses' source='mit_user' container='list'>
			<attribute name='houseNo' constant='123' />
		</part>
	</iterate>
</setup>
```

10 Generating XML Files

Benerator offers different options to generate XML files:

- DbUnitEntityExporter: A consumer writes any created entity to a file in DbUnit XML format. Use this if you only need a DbUnit file or want a simple, flat XML-based export for import in other tools. See the component reference for more information.
- XMLEntityExporter: A consumer which is not much more powerful than the DbUnitEntityExporter: It renders each simple-type entity attribute as an XML attribute and each sub entity as nested XML element. See the component reference for more information.
- Schema-based generation: An approach that uses an XML Schema file to automatically generate an arbitrary number of XML files. The schema files may be annotated with a similar syntax like used in Benerator descriptor files. This is the most powerful XML generation option:

10.1 Schema-based XML file generation

In this approach, an XML schema is used as the central descriptor file. Benerator is able to generate from a plain schema file automatically, but inserting XML schema annotations, you can configure test data generation almost as versatile as with the classic descriptor-file-based approach.

XML schema support is not yet fully implemented. The limitations are:

- No support for recursion of the same element type, e.g. categories containing other categories
- No support for mixed content. benerator is concerned with generation of data structures, while mixed-type documents generally apply for natural-language documents.
- groups are not supported
- sequences may not have maxOccurs > 1
- namespace support is only rudimentary, problems may arise on different types with equal names
- schema include is not supported yet
- ids and idrefs are not resolved automatically

If Benerator is your tool of choice and you need a feature urgently, please contact Volker Bergmann by E-Mail or forum.

10.1.1 Introduction

For the first trials, use a simple XML Schema file. We are beginning without annotations and save the following XML Schema with the name transactions.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ben="http://databene.org/benerator-0.8.1.xsd"
    xmlns="http://databene.org/shop-0.8.1.xsd"
    targetNamespace="http://databene.org/shop-0.8.1.xsd"
    elementFormDefault="qualified">

  <xs:element name="transactions">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="transaction" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="transaction">
    <xs:complexType>
      <xs:attribute name="id" type="xs:long" use="required" />
      <xs:attribute name="comment" type="xs:string" use="required" />
      <xs:attribute name="ean_code" type="xs:string" use="required" />
      <xs:attribute name="price" type="price-type" use="required" />
      <xs:attribute name="items" type="xs:integer" />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="price-type">
    <xs:restriction base="xs:decimal">
      <xs:minInclusive value="0"/>
      <xs:totalDigits value="8" />
      <xs:fractionDigits value="2" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

This defines an XML file format which has a <transactions> attribute as root element which contains an arbitrary number of <transaction> elements. Each <transaction> has the attributes 'id', 'comment', ean_code', 'price' and 'items'. The 'price-type' specifies decimal values with a total of 8 digits, 2 of which are decimal digits.

You can invoke XML file generation using Benerator directly or using the Maven Benerator Plugin. Let's call Benerator directly from the shell for now. Open a console, go to the directory which contains your schema file and invoke (under Windows):

```
createXML.bat transactions.xsd transactions tx-{0}.xml 2
```

On Unix systems type:

```
createXML.sh transactions.xsd transactions tx-{0}.xml 2
```

This tells Benerator to generate 2 xml files named 'tx-1.xml' and 'tx-2.xml' based on the schema file 'transactions.xsd' using the 'transactions' element as root element.

Open one of the generated files and you will see the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<transactions elementFormDefault="unqualified"/>
```

So, what did Benerator do wrong? Nothing, it is a perfectly schema-valid document. Since minOccurs of the transaction ref is zero, Benerator takes the easy choice: Remember: One of Benerator's strengths is to configure generation of valid data as easy as possible and in project stages as early as possible. With the

chosen approach, you need to spend less time for explicitly configuring element removal which are not yet supported by your application.

For configuring the generation of <transaction> elements, you need to add an annotation to your schema. The 'ref' configuration in the 'sequence' is the right place to configure cardinalities of included sub elements:

```
  <xs:element name="transactions">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="transaction">
          <xs:annotation>
            <xs:appinfo>
              <ben:part minCount="5" maxCount="10"/>
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Run Benerator again and you will notice, that Benerator generated files like this one, each containing 5 to 10 transactions:

```
<?xml version="1.0" encoding="UTF-8"?>
<transactions elementFormDefault="unqualified">
  <transaction id="4" comment="OBNBHQWMJYSPAHOCNBGDPGUXUQK"
      ean_code="KJCDPUJNK" price="1" items="6"/>
  <transaction id="6" comment="UIXSXLGFBIRP"
      ean_code="MW" price="3" items="7"/>
  <transaction id="4" comment="CRWGBGEKLRTZQADE"
      ean_code="MXIESHSXQVLFJIBC" price="5" items="5"/>
  <transaction id="9" comment="FVBABHSYXZJHQYCVCWJ"
      ean_code="FRPHJBOKUWHYKWHCWEIJBHVHIMV" price="1" items="9"/>
  <transaction id="9" comment="FZBNSLBEBZMTGPZJUG"
      ean_code="MNYYPRKM" price="7" items="5"/>
  <transaction id="7" comment="KIWPOOHNV"
      ean_code="CRXMHAGAC" price="3" items="7"/>
  <transaction id="9" comment="JETNYCMECHGUPSUKLKSEA"
      ean_code="ICY" price="1" items="5"/>
</transactions>
```

Now we have <transactions>, but their attribute values are not necessarily meaningful for our application. We need to configure attribute generation, too. Note however, that Benerator understands the definition of custom data types like the 'price-type' and automatically generates valid data, though taking the easy way of defaulting to integral numbers.

10.1.2 Configuring Attribute Generation

Now it is time to configure the attribute details. Let us start by declaring the 'id' attribute as ID

```
<xs:attribute name="id" type="xs:long" use="required">
  <xs:annotation>
    <xs:appinfo>
      <ben:id/>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
```

BTW: The XML Schema type ID is not yet handled automatically. You need to add an explicit <ben:id/> annotation for generating unique identifiers of the desired type.

Shorter random comments are generated based on a regular expression:

```
<xs:attribute name="comment" type="xs:string" use="required">
  <xs:annotation>
    <xs:appinfo>
      <ben:attribute pattern="[A-Z][a-z]{5,12}"/>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
```

You can configure number generation for the 'items' attribute by setting min, max, resolution and distribution values:

```
<xs:attribute name="items">
  <xs:annotation>
    <xs:appinfo>
      <ben:attribute type="short" min="1" max="27" distribution="cumulated"/>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
```

This makes benerator create 'items' numbers from 1 to 27 with a cumulated distribution which has its maximum at 14.

10.1.3 Using <variables> in XML Schema

Now for more complex data generation: You can use <variables> like in descriptor files. They need to be placed inside an <element>. Let us, for example, use a CSV file with product definitions, containing EAN and price for each article. First, the variable declaration:

```
<xs:element name="transaction">
  <xs:annotation>
    <xs:appinfo>
      <variable name="product" source="products.ent.csv" distribution="random"/>
```

```
    </xs:appinfo>
  </xs:annotation>
  ...
```

For each generation of a transaction, the <variable> is called to generated a new helper object, in this case providing a CSV data line with product data. The contents of this data are mapped using script expressions:

```
  ...
  <xs:attribute name="ean_code" type="xs:string" use="required">
    <xs:annotation>
      <xs:appinfo>
        <ben:attribute script="product.ean_code"/>
      </xs:appinfo>
    </xs:annotation>
  </xs:attribute>

  <xs:attribute name="price" type="price-type" use="required">
    <xs:annotation>
      <xs:appinfo>
        <ben:attribute script="product.price"/>
      </xs:appinfo>
    </xs:annotation>
  </xs:attribute>
```

Using a CSV file with product definitions:

```
ean_code,name,category_id,price,manufacturer
8000353006386,Limoncello Liqueur,DRNK/ALCO,9.85,Luxardo
3068320018430,Evian 1.0 l,DRNK/SOFT,1.95,Danone
8076800000085,le Lasagnette,FOOD/MISC,0.89,Barilla
7610400071680,Connaisseurs,FOOD/CONF,16.95,Lindt
```

we finally get a satisfactory result:

```
<?xml version="1.0" encoding="UTF-8"?>
<transactions elementFormDefault="unqualified">
  <transaction id="1" comment="Sczexyozcpc"
      ean_code="8076800000085" price="0.89" items="9"/>
  <transaction id="2" comment="Nglija"
      ean_code="8000353006386" price="9.85" items="11"/>
  <transaction id="3" comment="Miejztqhgaoc"
      ean_code="7610400071680" price="16.95" items="14"/>
  <transaction id="4" comment="Tzoxzrupygjfi"
      ean_code="8000353006386" price="9.85" items="11"/>
```

```
  <transaction id="5" comment="Sufqdrku"
      ean_code="3068320018430" price="1.95" items="17"/>

  <transaction id="6" comment="Jbtqsft"
      ean_code="8000353006386" price="9.85" items="14"/>

  <transaction id="7" comment="Lvivruwwxmay"
      ean_code="8076800000085" price="0.89" items="10"/>

</transactions>
```

You might as well want to calculate the total price. You can easily do so using a script expression, e.g. script="this.price * this.items". Note that the elements are evaluated and generated in the order in which they are declared, so the 'total sum' field must be defined after the used terms 'price' and 'items'.

10.1.4 Importing Properties File Data

You can import settings from properties files by placing <include>s in the schema's root node annotation:

```
<xs:schema ...>

      <xs:annotation>

            <xs:appinfo><ben:include uri="benerator.properties"/></xs:appinfo>

      </xs:annotation>
```

10.2 Generating XML in classic descriptor files

Generating data from an XML schema file is somewhat limited. Alternatively, you can use the classic Benerator descriptor files to generate entity data and write it to XML with a special consumer. If you do not need to adhere to a predefined XML schema, but simply want some XML for easy postprocessing, you might get what you need, if you use the XMLEntityExporter or the even simple DbUnitEntityExporter. Future Benerator versions will provide better options.

10.2.1 Using data types from XML schema files

Including an XML schema in a classic descriptor file makes its data types available for explicit data generation:

```
<include uri="shop.xsd" />

<generate type="product" count="5" consumer="ConsoleExporter"/>
```

10.3 Conclusion

Almost the full feature set of Benerator descriptor files is available for XML Schema-based file generation. If you know the Benerator descriptor file syntax, it is a straightforward and relatively simple process to annotate descriptor files. However, if you just need to export XML-formatted data and write an own XML parser for importing the data somewhere else, you might prefer to use the DbUnitEntityExporter (flat structure) or XMLEntityExporter (hierarchical structure), possibly in combination with an XSL transformation.

11 Advanced Topics

11.1 JavaBeans and the Benerator Context

You can instantiate JavaBeans by an intuitive syntax like this:

```
<bean id="helper" class="com.my.Helper">
  <property name="min" value="5"/>
  <property name="max" value="23"/>
</bean>
```

The *class* attribute denotes which JavaBean class to instantiate (by the default constructor). The enclosed property tags cause the JavaBean's properties and attributes to be set to appropriate values. Benerator converts common types automatically. If benerator cannot perform conversion of a custom type, you can define a custom ConverterManager setup (see databene-commons). Date and time formatting is supported according to ISO 8601 Conventions.

Objects are made available by exposing them in a context. The id attribute defines the name with which an object can be found, e.g. for a 'source' or 'ref' attribute of another element's setup.

So the example above creates an instance of a DBSystem JavaBean class, setting its properties to values for connecting a database. The object is retrievable by the context with the id 'db'.

Note: The class DBSystem implements the interface 'System' which provides (among other features) meta information about the entities (tables) contained in the database.

You can create references to other objects declared before by a 'ref'-attribute in the bean declaration. The following example shows this for a task setup, but this can be applied to beans and consumers as well.

Note: You may implement the System interface for connecting to other system types like SAP or Siebel systems.

JavaBeans may refer each other (see proxy.target) and may have collection or attribute properties (see log_csv.components) as shown in the following example:

```
<bean id="csv" class="CSVEntityExporter">
  <property name="uri" value="customers.csv"/>
  <property name="columns" value="salutation,first_name,last_name"/>
</bean>

<bean id="proxy" class="shop.MyProxy">
  <property name="target" ref="csv"/>
</bean>

<bean id="log_csv" class="ConsumerChain">
  <property name="components">
    <bean class="LoggingConsumer"/>
    <idref bean="proxy"/>
  </property>
</bean>
```

You can use all inline construction styles in a spec attribute, too. See ???, ??? and ???

You can invoke methods on beans using DatabeneScript:

```
<execute>myBean.init(47 + 11)</execute>
```

11.2 Importing Java classes

benerator provides an import facility similar to the Java language. You can import classes, packages and domains.

So, instead of using the fully qualified name:

```
<bean id="special" class="com.my.SpecialGenerator" />
```

you can import the class and use the local name:

```
<import class="com.my.SpecialGenerator" />

<bean id="special" class="com.my.SpecialGenerator" />
```

The following alternative works as well and imports all classes of the com.my package:

```
<import class="com.my.*" />

<bean id="special" class="com.my.SpecialGenerator" />
```

Domains can be imported as well. For the built-in benerator domains, only the domain name is necessary, for custom domains, the fully qualified name of the domain's top level package. For a built-in domain:

```
<import domains="person" />

<bean id="personGen" class="PersonGenerator" />
```

For a custom domain:

```
<import domain="com.my" />

<bean id="myGen" class="MyGenerator" />
```

11.3 Looking up services/objects via JNDI

Benerator provides an InitialContext class in the JNDI platform package. It can be used to locate objects and make them available in Benerator:

```
<import platforms="jndi" />

<bean id="ctx" class="InitialContext">

    <property name="factory" value="..." />

    <property name="url" value="..." />

    <property name="user" value="..." />

    <property name="password" value="..." />

</bean>

<bean id="ejb" spec="ctx.lookup('cons')" />
```

11.4 Calculating global sums

Sometimes you need to calculate the total sum of one field over all generated instances, e.g. for calculating checksums. The AddingConsumer is there to help you with this task. Instantiate it as <bean>, specifying the field name to add and the number type to use, then use it as consumer and finally query its 'sum' property value with a script expression:

```
<bean id="adder" spec="new AddingConsumer('txn_amount', 'long')" />

<generate type="deb_transactions" count="100"
        consumer="ConsoleExporter, adder">

    <attribute name="txn_amount" type="long"
        min="1" max="100" distribution="random" />

</generate>

<generate type="trailer_record" count="1" consumer="ConsoleExporter">
```

```
      <attribute name="total" script="adder.sum" />
</generate>
```

11.5 Querying information from a system

Arbitrary information may be queried from a system by a 'selector' attribute, which is system-dependent. For a database SQL is used:

```
<generate type="db_order" count="30" pageSize="100" consumer="db">
    <attribute name="customer_id" source="db"
        selector="select id from db_customer" cyclic="true"/>
</generate>
```

You can use script expressions in your selectors, e.g.

```
selector="{ftl:select ean_code from db_product where country='${country}'}"
```

The script is resolved immediately before the first generation and then reused. If you need dynamic queries, that are re-evaluated, you can specify them with double brackets:

```
selector="{{ftl:select ean_code from db_product where country='$
{shop.country}'}}"
```

Example:

```
<generate type="shop" count="10">
  <attribute name="country" values="DE,AT,CH"/>
  <generate type="product" count="100" consumer="db">
    <attribute name="ean_code" source="db"
      selector="ftl:selector="{{select ean_code from db_product
        where country='${shop.country}'}}"/>
  </generate>
</generate>
```

11.6 The MemStore

The MemStore is a simple implementation of the StorageSystem interface which allows yo to store entity data in RAM and perform simple queries on them. This is useful in cases where you need to generate data where there are internal dependencies but no database is involved, e.g. generating an XML file with products and orders.

A MemStore is declared with a <memstore> element:

```
<memstore id="store"/>
```

You can use it as a consumer for storing data (in this case products):

```
<generate type="product" count="10" consumer="store,ConsoleExporter">
      <id name="id" type="int" />
      <attribute name="name" pattern="[A-Z][a-z]{4,12}" />
</generate>
```

Afterwards you can query the generated products for referencing them in generated orders:

```
<generate type="order" count="10" consumer="ConsoleExporter">
  <variable name="product" source="store" type="product" />
  <id name="id" type="int" />
```

```
  <attribute name="product_id" script="product.id"/>
</generate>
```

Note that you can only query for entities – if you need only an attribute of an entity, you must first use a variable to get the entity and then a script to get the required attribute.

You can use a distribution:

```
<variable name="product" source="store" type="product" distribution="random"/>
```

A simple form of query is supported by a 'selector' element. Its content must be a script expression that serves as a filter. The expression is consecutively evaluated on each element (as candidate) and has to return true, if the candidate is accepted, otherwise false. The script can access each candidate with the keyword `_candidate`.

As an example, here is a query which only returns products whose name starts with 'A':

```
<variable name="product" source="store" type="product"
    selector="_candidate.name.startsWith('A')"/>
```

11.7 Datasets

You can define datasets and combine them to supersets. This mechanism lets you also define parallel and overlapping hierarchies of nested datasets.

Definition of a hierarchies is separated from the definition of dataset values for a concrete topic. So you can define a dataset grouping for regions, mapping continents, countries, states and departments and apply this grouping to define and combine sets of e.g. cities, person names or products.

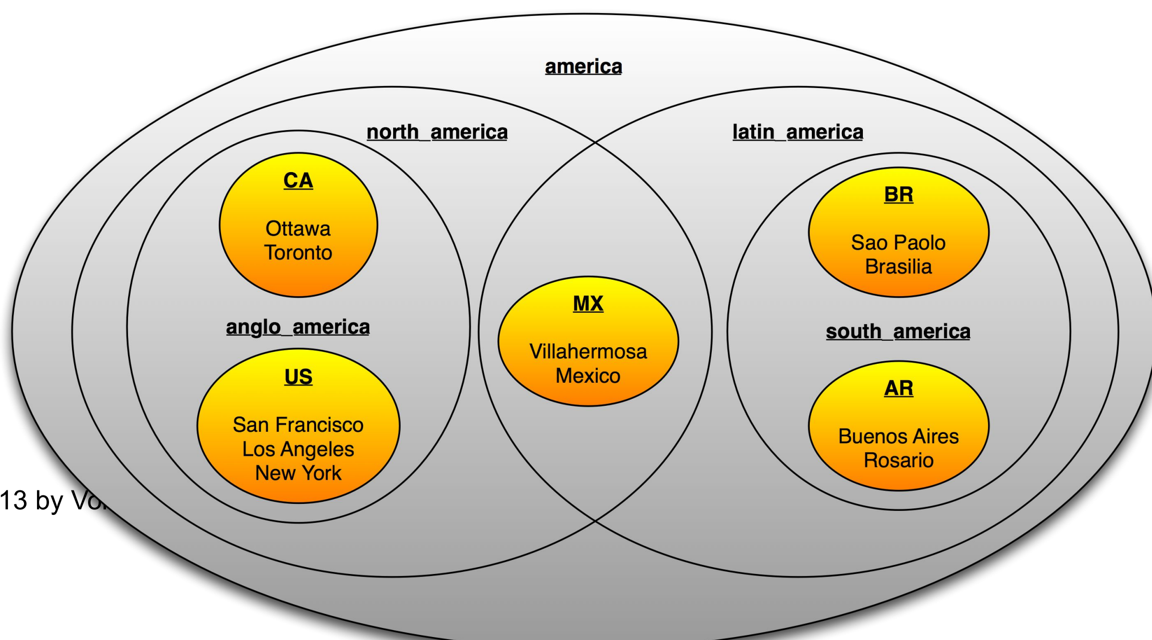
We will apply the mechanism here for cities in geographical regions. You can find the example files in the distribution's directory demo/dataset/

A dataset is identified by a code. For a country, its ISO code is an appropriate choice, but you are free to define and choose what is useful for your application.

Assume you wanted to process some American countries: US (USA), CA (Canada), MX (Mexico), BR (Brazil), AR (Argentina)

You could group them geographically (North America vs. South America) or by language (Latin America vs. Anglo America). You could do both in parallel by defining area sets in a file area.set.properties :

```
latin_america=MX,BR,AR
anglo_america=US,CA
north_america=US,CA,MX
south_america=BR,AR
america=north_america,south_america
```



The simple datasets would be defined in CSV files:

cities_US.csv:

```
San Francisco
Los Angeles
New York
```

cities_CA.csv:

```
Ottawa
Toronto
```

cities_MX.csv

```
Mexico
Villahermosa
```

cities_BR.csv

```
Sao Pãolo
Brasilia
```

cities_AR.csv

```
Buenos Aires
Rosario
```

You can now use this setup to generate city names for any of the specified regions. For North American cities you could specify

```
<echo message="north american cities:" />
<generate type="city" consumer="exporter" count="10">
    <attribute name="name" unique="true"
        source="city_{0}.csv" encoding="UTF-8"
        dataset="north_america" nesting="area" />
</generate>
```

and generate the output:

```
north american cities:
city[name=Mexico]
city[name=Los Angeles]
city[name=San Francisco]
city[name=New York]
city[name=Villahermosa]
```

```
city[name=Ottawa]
city[name=Toronto]
```

11.7.1 Region nesting

benerator comes with a predefined nesting definition for geographical datasets, called 'region':

```
europe=western_europe,central_europe,\
    southern_europe,eastern_europe,northern_europe
western_europe=PT,ES,AD,FR,MC
central_europe=BE,NL,LU,DE,CH,AT,LI
southern_europe=IT,SM,GR,CY,TR
eastern_europe=AL,SI,CZ,HU,PL,RU,RO,BG,HR,BA,EE,LT,LV,SK,UA
northern_europe=UK,IE,DK,SE,NO,FI,IS
near_east=AF,IR,IL,JO,KZ,PK,QA,SA,AE
africa=EG,GH,KE,ZA
north_america=US,CA
central_america=MX,BS
america=north_america,central_america,south_america
south_america=AR,BR,EC
asia=JP,IN,ID,KR,KP,MY,SG,TW,TH
australia=AU,NZ
```

11.8 Chaining generators

Generators can be chained, composed, or reused in different contexts. You can do so by instantiating a generator as JavaBean and referring it in properties of other JavaBean-instantiated generators or specifying it as 'source' attribute like an importer.

```
<!-- creates a text generator →
<bean id="textGen" class="RegexStringGenerator">
    <property name="pattern" value="([a-z]{3,8}[ ])*[a-z]{3,8}\."/>
</bean>


<!-- wraps the text generator and creates messages -->
<generate type="message" count="10" consumer="LoggingConsumer">
    <attribute name="text" source="textGen"
        converter="MessageConverter" pattern="Message: ''{0}''"/>
</generate>
```

11.9 Invoking Benerator programmatically

For integrating Benerator with other applications, you can invoke it programmatically in a Java application too.

11.9.1 Invoking Benerator as a whole

For executing descriptor files do the following:

```
// create an instance of the Descriptor runner specifying the descriptor file
DescriptorRunner runner = new DescriptorRunner("path/to/file/benerator.xml");
BeneratorContext context = runner.getContext();
// use the BeneratorContext to set locale, file encoding, ... as you need
context.setValidate(false);
runner.run();
```

11.9.2 Making Benerator generate data and invoke a custom class with it

If you want to use Benerator for feeding a custom class with generated data, implement the Consumer interface in a way that connects to your program, instantiate it and register it with the BeneratorContext:

```
DescriptorRunner runner = new DescriptorRunner("path/to/file/benerator.xml");
BeneratorContext context = runner.getContext();
context.setValidate(false);
MyConsumer myConsumer = new MyConsumer();
context.set("myConsumer", myConsumer); // add a custom Consumer
runner.run();
```

A simplistic implementation could simply write entities to the console, e.g.

```
class MyConsumer extends AbstractConsumer<Entity> {
    List<Entity> products = new ArrayList<Entity>();
    public void startConsuming(Entity entity) {
        products.add(entity);
    }
}
```

11.9.3 Using generators defined in a descriptor file

You can define data generation in a Benerator descriptor file, make Benerator configure the generator and hand it out to your application.

First, define a descriptor file, e.g.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<setup xmlns="http://databene.org/benerator/0.8.1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://databene.org/benerator/0.8.1
          http://databene.org/benerator-0.8.1.xsd">
    <generate type="address">
        <attribute name="city" values="'Munich', 'New York', 'Tokyo'" />
    </generate>
</setup>
```

Then you can get the address generator from Benerator by calling:

```
BeneratorContext context = new BeneratorContext();
Generator<?> generator = new DescriptorBasedGenerator("benerator.xml",
    "address", context);
generator.init(context);
ProductWrapper wrapper = new ProductWrapper();
for (int i = 0; i < 10; i++)
    System.out.println(generator.generate(wrapper));
generator.close();
```

11.10 Tasks

In many cases, data generation based on the descriptor file format may be too complex and a script based generation too slow.

In these cases you can provide a custom class that implements the org.databene.task.Task interface and can be invoked from the benerator descriptor file. For example you could write a Task implementation that calls business logic for triggering complex operations on business objects.

Such a class can be instantiated and invoked with a similar syntax like any JavaBean, e.g.:

```
<run-task class="com.my.SpecialTask">
  <property name="uri" value="base.csv"/>
  <property name="db" ref="db"/>
</run-task>
```

You can instantiate and initialize a task like a <bean> (see *JavaBeans and the benerator Context*).
Tasks can be executed in pages, ensuring that the total number of execution over all threads is the page size. For this, The element run-task also supports the attributes

- count : the total number of times the Task is executed (defaults to 1)
- pageSize : the number of invocations to execute together (defaults to 1), e.g. for grouping pageSize invocations to a single transaction.
- pager: injects a class (implementing the PageListener interface) to perform custom paging operations (like flushing data to file or database)

```
<run-task class="my.SpecialTask" count="1000" pageSize="100"
        pager="my.PagingStrategy">
    <property name="message" value="I'm special"/>
</run-task>
```

11.11 Staging

Combining scripting and property files, you get a staging mechanism. A typical example would be to use different setups for developing data generation on a local system and using it to produce mass data on a test environment. The basic approach is to extract variable properties to properties files, e.g. development.properties:

```
user_count=100
```

```
db_batch=false
pageSize=1
db_uri=jdbc:oracle:thin:@10.37.129.4:1521:XE
db_driver=oracle.jdbc.driver.OracleDriver
db_user={user.name}
db_password={user.name}
db_schema={user.name}
```

and perftest.properties:

```
user_count=1000000
db_batch=true
pageSize=1000
db_uri=jdbc:oracle:thin:@134.53.26.183:1521:MYAPP
db_driver=oracle.jdbc.driver.OracleDriver
db_user=myapp
db_password=myapp
db_schema=myapp
```

You can then decide which configuration to use by setting a stage setting as Java virtual machine parameter, e.g. write

```
<mvn benerator:generate -Dstage=development/>
```

11.12 Template data structures

You can use DbUnit import files for replicating entity graph structures many times on each generated object. Say, for each customer in a tested online shop, a default order structure should be created. You would then define the order structure in a DbUnit file.

```
<dataset>
  <db_order_item
    order_id="{db_order.id}"
    number_of_items="2"
    product_ean_code="8076800195057"
    total_price="2.40" />
  <db_order_item
    order_id="{db_order.id}"
    number_of_items="1"
    product_ean_code="8006550301040"
    total_price="8.70" />
</dataset>
```

and then create an order for each customer that imports its sub structure from the DbUnit file:

```
<generate type="db_order" consumer="db">
  <id name="id" />
  <reference name="customer_id"/>
  <iterate type="db_order_item"
```

```
      source="demo/shop/default_order.dbunit.xml"
      consumer="db">
    <id name="id" />
    <reference name="order_fk" script="db_order.id" />
  </iterate>
</generate>
```

Of course, you have to care for appropriate ids yourself.

11.13 Generating arrays

Instead of entity data, you can generate arrays using type="array" for declaring the generated type and <value> elements for configuring the generated components:

```
  <generate type="array" count="5" consumer="ConsoleExporter">
    <value type="string" pattern="[A-Z]{5}" />
    <value type="int" min="1" max="42" />
  </generate>
```

This is useful for special consumers that require array types, e.g. for generating parameters for the JavaInvoker which calls Java methods.

11.14 Scoped Generation

All nested generator components have a scope which influences its life cycle: When generation in a scope is finished, its sub components are reset before the next invocation.

11.14.1 Default (implicit) scope

As an example, assume you are generating data for a shop application which takes orders composed of order items. Each order and order item must have its unique id, but the order_item should have a sequence number which represents its position number within its associated order:

```
<generate name="the_order" type="order" count="2" consumer="ConsoleExporter">
      <id name="id" type="int" generator="IncrementalIdGenerator" />
      <generate name="the_item" type="order_item" minCount="1" maxCount="3"
                  consumer="ConsoleExporter">
            <id name="id" type="int" generator="IncrementalIdGenerator" />
            <reference name="order_id" script="the_order.id" />
            <attribute name="item_no" type="int" distribution="increment"/>
            <attribute name="product" type="string" pattern="[A-Z][a-z]{3,8}"/>
            <attribute name="count" type="int" min="1" max="3" />
      </generate>
</generate>
```

When running this descriptor, the output is (after indenting and formatting):

```
order[id=1]
      order_item[id=1, order_id=1, item_no=1, product=Pgbxll, count=2]
      order_item[id=2, order_id=1, item_no=2, product=Pmce, count=1]
order[id=2]
      order_item[id=3, order_id=2, item_no=1, product=Entlkzkjv, count=2]
      order_item[id=4, order_id=2, item_no=2, product=Jgqp, count=3]
```

This exhibits the behaviour with default (implicit) scope, which is tied to the owner of a component. In this

case, the scope of the item_no component is the order_item entity: When one generation loop of „the_item“ is through, the contained component generators are reset before running the next loop (belonging to a new instance of 'the_order').

11.14.2 Explicit scope

To understand explicit scoping, let us first extend the previous example, assuming that each single product instance ordered is shipped in an own package, which shall be tracked. Thus, you would add a sub <generate> for the packages:

```
<generate name="the_order" type="order" count="2" consumer="ConsoleExporter">
      <id name="id" type="int" generator="IncrementalIdGenerator" />

      <generate name="the_item" type="order_item" minCount="1" maxCount="3"
                  consumer="ConsoleExporter">
            <id name="id" type="int" generator="IncrementalIdGenerator" />
            <reference name="order_id" script="the_order.id" />
            <attribute name="item_no" type="int" distribution="increment"/>
            <attribute name="product" type="string" pattern="[A-Z][a-z]{3,8}"/>
            <attribute name="count" type="int" min="1" max="3" />

            <generate type="package" count="{the_item.count}"
                        consumer="ConsoleExporter">
                  <id name="id" type="int" generator="IncrementalIdGenerator" />
                  <reference name="order_id" script="the_order.id" />
                  <attribute name="order_pkg_no" type="int"
                        distribution="increment" />
            </generate>

      </generate>

</generate>
```

Running this yields the output:

```
order[id=1]
      order_item[id=1, order_id=1, item_no=1, product=Pgbxll, count=2]
            package[id=1, order_id=1, order_pkg_no=1]
            package[id=2, order_id=1, order_pkg_no=2]
      order_item[id=2, order_id=1, item_no=2, product=Pmce, count=1]
            package[id=3, order_id=1, order_pkg_no=1]

order[id=2]
      order_item[id=3, order_id=2, item_no=1, product=Entlkzkjv, count=2]
            package[id=4, order_id=2, order_pkg_no=1]
      order_item[id=4, order_id=2, item_no=2, product=Jgqp, count=3]
            package[id=5, order_id=2, order_pkg_no=1]
            package[id=6, order_id=2, order_pkg_no=2]
```

OK, we see that this adds <count> packages to each order_item and the order_pkg_no numbering begins with 1 for each order_item..

Now suppose that the package number should be a sequence which is unique for a complete order, not just an order_item – this is where the package number generation needs a different scope, the one of the order, telling Benerator to apply the same lifecycle to the packages as to the components of the order:

```
<generate name="the_order" type="order" count="2" consumer="ConsoleExporter">
      <id name="id" type="int" generator="IncrementalIdGenerator" />

      <generate name="the_item" type="order_item" minCount="1" maxCount="3"
                  consumer="ConsoleExporter">
            <id name="id" type="int" generator="IncrementalIdGenerator" />
```

```
            <reference name="order_id" script="the_order.id" />
            <attribute name="item_no" type="int" distribution="increment"/>
            <attribute name="product" type="string" pattern="[A-Z][a-z]{3,8}"/>
            <attribute name="count" type="int" min="1" max="3" />

            <generate type="package" count="{the_item.count}"
                        consumer="ConsoleExporter">
                  <id name="id" type="int" generator="IncrementalIdGenerator" />
                  <reference name="order_id" script="the_order.id" />
                  <attribute name="order_pkg_no" type="int"
                        distribution="increment" scope="the_order"/>
            </generate>

      </generate>

</generate>
```

yielding the desired result:

```
order[id=1]
      order_item[id=1, order_id=1, item_no=1, product=Pgbxll, count=2]
            package[id=1, order_id=1, order_pkg_no=1]
            package[id=2, order_id=1, order_pkg_no=2]
      order_item[id=2, order_id=1, item_no=2, product=Pmce, count=1]
            package[id=3, order_id=1, order_pkg_no=3]

order[id=2]
      order_item[id=3, order_id=2, item_no=1, product=Entlkzkjv, count=2]
            package[id=4, order_id=2, order_pkg_no=1]
      order_item[id=4, order_id=2, item_no=2, product=Jgqp, count=3]
            package[id=5, order_id=2, order_pkg_no=2]
            package[id=6, order_id=2, order_pkg_no=3]
```

11.14.3 Global scope

Using scope="/", any component can be configured to have global scope which means that it is never reset. This can also be used as a performance improvement for resource-heavy database queries combined with a caching distribution.

11.15 Composite Data Generation

For databases, flat data generation is used: Each generated entity stands for itself and is persisted without a context. Nesting <generate> elements only corresponds to a concept of loop and sub loop, not to a hierarchical nesting of data. First the top level element is created and sent to its consumer(s) then its child elements are generated and sent to their consumer(s). It is up to the consumer(s) to create and track context and interpret the generated data in a hierarchical manner.

In cases of intrinsically hierarchical data (mongoDB or XML), a root data entity needs to be generated in a composite manner and processed differently: All nested elements are generated until the full tree is built and then the root element is sent to the consumer (and only this element). The consumer can then scan through the entity tree and do its job.

This is what the <part> element has been introduced for. It takes the same parameters as the <generate> element, but does not call a consumer with the generated data, but puts them into a collection which is set as the parent entity's property.

For example,

```
<generate type='user' minCount='100' consumer='db'>
      <attribute name='name' type='string' />
```

```
	<attribute name='age' type='int' min='18' max='78' />
	<part name='addresses' container='list' count='2'>
		<attribute name='street' pattern='[A-Z][a-z]{4} Street'/>
		<attribute name='houseNo' type='int' min='2' max='9' />
	</part>
```

generates 100 user entities, of which each one has an 'address' component, which is a list of 2 address entities.

Supported settings for the part's container attribute are:

- list
- set
- array

The default is 'list'.

11.16 Composite Data Iteration

Using a data source which provides hierarchical data, nested collection components (like user.addresses in the previous example) can be iterated explicitly, specifying the root component as **source** and providing the component **name**.

Here is an example which performs a primitive data anonymization job on users and their addresses in a mongoDB server:

```
<setup>
	<import platforms='mongodb' />
	<mongodb id='db' environment='mymongo' />
	<iterate type='user' source='db'
			consumer='db.updater(),ConsoleExporter'>
		<attribute name='age' constant='33' />
		<part name='addresses' source='user' container='list'>
			<attribute name='houseNo' constant='123' />
		</part>
	</iterate>
</setup>
```

12 Generating Unique Data

12.1 ID Generation

For generating unique data for dataset identifiers like primary keys in a database, see the chapters „Common ID Generators“ for a complete ID generator reference and „Using databases“ for database-related id generators.

12.2 Unique Number Generation

Most → Sequences are able to generate unique numbers. Just apply a unique="true" to the number configuration:

```
<attribute name="n" type="int" min="3" max="99" unique="true" />
```

12.3 Unique String Generation

12.3.1 Uniqueness with Regular Expression

There are ID generators which generate UUID strings, but in most cases you have constraints on string length and character select which require you to have something more individual and configurable.

One of the best general approches is to use Benerator's feature to generate unique strings that match a regular expression. For example, for generating unique phone numbers, you could write:

```
<attribute name="phone" type="string"
      pattern="[1-9][0-9]{2}\-[0-9]{4}\-[0-9]{5}" unique="true" />
```

For an introduction to regular expressions, read the chapter „Regular Expressions“.

12.3.2 Making Strings unique

Sometimes you have less strict constraints on the strings you want to make unique. A good example is a common derivation of user names from their real names which takes the first letter of the first name and appends the last name. This might lead to non-unique results, since John Smith and Joshua Smith would get the same user name jsmith. The usual solution is to append a number to make the string for the second jsmith unique again: jsmith2. This is exactly, what the UniqueStringConverter does:

```
<generate type="user" count="10" consumer="ConsoleExporter" />

   <variable name="person" generator="PersonGenerator"/>

   <attribute name="user_name"
      script="person.givenName.substring(0, 1) + person.lastName"
      converter="ToLowerCaseConverter, UniqueStringConverter"/>

</generate>
```

Note: The UniqueStringConverter keeps all used strings in memory, so when generating some billion strings, you might get memory problems.

12.4 Removing Duplicate Values

If you need a more individual generation algorithm of which you do not know (or care) how to make it unique, you can append a UniqueValidator to filter out duplicate values.

```
<attribute name="code" pattern="[A-Z]{6,12}" validator="UniqueValidator"/>
```

Note: The UniqueValidator keeps all used strings in memory, so when generating some billion strings, you might get memory problems.

12.5 Unique iteration through a source

When iterating data from a data source and requiring uniqueness, you need to assure for yourself, that the source data is unique:

```
<attribute name="code" type="string" source="codes.csv" />
```

When applying a distribution to the iterated data, configure unique="true" for assuring that the distribution does not repeat itself:

```
<attribute name="code" type="string" source="codes.csv"
      distribution="random" unique="true" />
```

12.6 Unique Generation of Composite Keys

As an example, let's have a look the following code:

```
<generate type="product" count="6" consumer="ConsoleExporter">

   <attribute name="key1" type="int" />
   <attribute name="key2" type="int" />

</generate>
```

If we need to generate unique combinations of key1 and key2 we have differrent alternatives:

12.6.1 Simplistic Approach

If each value is unique, the combination of them is unique too. The following setting:

```
<generate type="product" count="6" consumer="ConsoleExporter">

   <attribute name="key1" type="int" distribution="increment" unique="true" />
   <attribute name="key2" type="int" distribution="increment" unique="true" />

</generate>
```

The generated values are:

```
product[key1=1, key2=1]

product[key1=2, key2=2]

product[key1=3, key2=3]

product[key1=4, key2=4]

product[key1=5, key2=5]

product[key1=6, key2=6]
```

12.6.2 Cartesian Product

For generating unique composite keys the most convenient way is to create a cartesian product of unique components by nesting two creation loops.

For making the combination of key1 and key2 in the following descriptor unique:

one would add an outer 'dummy' loop and create helper variables x and y in a way that they can be combined like in a cartesian product:

```
<generate type="dummy" count="2"> <!-- no consumer! -->

   <variable name="x" type="int" distribution="increment" unique="true" />

   <generate type="product" count="3" consumer="ConsoleExporter">

      <variable name="y" type="int" distribution="increment" unique="true" />

      <attribute name="key1" type="int" script="x"/>
```

```
      <attribute name="key2" type="int" script="y"/>
   </generate>
</generate>
```

The generated values are:

```
product[key1=1, key2=1]
product[key1=1, key2=2]
product[key1=1, key2=3]
product[key1=2, key2=1]
product[key1=2, key2=2]
product[key1=2, key2=3]
```

12.6.3 Prototype Approach

You can use the prototype approach for getting unique compsite keys: A variable's generation algorithm needs to assure uniqueness of the combination:

```
<generate type="product" count="6" consumer="ConsoleExporter">
   <variable name="p" generator="my.HelperClass"/>
   <attribute name="key1" type="int" script="p.x" />
   <attribute name="key2" type="int" script="p.y" />
</generate>
```

The most frequent application of this approach is the generation of unique database references using a prototype query. See the chapter Prototype Queries.

12.7 Achieving local uniqueness

Sometimes values need to have uniqueness of an identity component of a 'child' entity only in the context of another ('parent') item.

One simple solution is of course to have it globally unique.

If there are more constraints involved, you can of course use an appropriate generator but need to nest the generation of parent and child:

```
<generate type="parent" count="5" consumer="ConsoleExporter">
   <generate type="product" count="5" consumer="ConsoleExporter">
      <variable name="y" type="int" distribution="increment" unique="true">
      <attribute name="key1" type="int" script="x"/>
      <attribute name="key2" type="int" script="y"/>
</generate>
```

13 Scripting

Benerator supports arbitrary scripting languages that are supported by Java and has an own scripting language DatabeneScript which is designed specifically for the purpose of data generation.

The invocation syntax is as described for SQL invocation and inlining: You can include the script inline like this:

```
<database id="db" url="jdbc:hsqldb:hsql://localhost:9001"
        driver="org.hsqldb.jdbcDriver" schema="public" user="sa"/>


<execute type="js">
        importPackage(org.databene.model.data);
        print('DB-URL' + db.getUrl());
        // create user Alice
        var alice = new Entity('db_user');
        alice.set('id', 1);
        alice.set('name', 'Alice');
        db.store(alice);


        // create user Bob
        var bob = new Entity('db_user', 'id', '2', 'name', 'Bob');
        db.store(bob);


        // persist everything
        db.flush();
</execute>
```

As you see with the db variable, all objects of the benerator context are made provided to the script. In this case, it is a DBSystem bean, which is used to store Entities created by the script. So, you can import objects of arbitrary Java classes and use them in your favorite scripting language.

Alternatively to inlining script text, you can put it in a script file and invoke this:

```
<execute uri="test.js" />
```

You can bind a language of choice by using the mechanisms of JSR 223: Scripting for the Java Platform.

With Java 6 for Windows, a JavaScript implementation is shipped. For all other platforms and languages you need to configure language support.

The following attributes are available for the <execute> element:

- uri: the URI of the script file to execute
- encoding: the encoding of the script file
- type: Type (language) of the script
- target: a target to execute the script on, typically a database for a SQL script
- onError: How to handle errors. One of (ignore, trace, debug, info, warn, error, fatal. fatal causes benerator to cancel execution in case of an error.
- optimize: boolean flag that tells benerator whether it may optimize script execution for the sake of performance. E.g. For an Oracle SQL script this would leave out comments for faster table creation.

benerator supports the following script types:

- shell: system shell invocations, e.g. for invoking batch files.
- sql: SQL, it requires specification of the database in a target property.
- jar: java library files with a configured main-class
- ben: DatabeneScript, which is the default script language
- ftl: FreeMarker
- JSR 223: Any language that has been plugged into the local Java environment, e.g. JavaScript, JRuby, Jython, Groovy and many more.

13.1 Shell scripting

You can call shell files or issue shell commands. When inlining shell commands, script expressions will be resolved. So you could, for example, use global properties for setting parameters of a sqlplus call:

```
<execute type="shell">{ftl:sqlplus ${dbUser}/${dbPassword}@${database}
@create_tables.sql}</execute>
```

14 DatabeneScript

14.1 Motivation

DatabeneScript is a script language designed for simplifying test data generation. Text-targeted template languages like FreeMarker and general languages like JavaScript have specific problems, which can be avoided by a language tailored to the task of data generation.

DatabeneScript uses many elements familiar from C-style languages like Java and C#, but adds some specials:

- More convenient object construction
- intuitive date arithmetic
- seamless integration and interaction with Benerator

14.2 Examples

14.2.1 Variable definition

```
<execute>x = 3</execute>
```

14.2.2 Variable assignment

```
<execute>x = x + 1</execute>
```

14.2.3 Object construction

Constructor-based:

```
new MyGenerator('alpha.txt', 23)
```

Properties-based:

```
new MyGenerator { filename='alpha.txt', limit=23 }
```

14.2.4 Date arithmetic

```
new Date() + 5000
```

14.2.5 Java integration

```
(org.databene.commons.SystemInfo.isWindows() ? 'win' : 'other')
```

Benerator integration

```
(context.contains('key') ? 'def' : 'none')
```

14.3 Syntax

14.3.1 Comments

Line comments start with // and include the rest of the line.

Normal comments begin with /* and end with */

14.3.2 White Space

Spaces, tabs, CR, LF, \u000C

14.3.3 Data Types

Signed integral numbers: long, int, short, byte, big_integer

Signed floating point numbers: double, float, big_decimal

Alpha: string, char

Date/time: date, timestamp

Other: boolean, object (Java object), binary (byte[])

14.3.4 Identifiers

The first character must be an ASCII letter or underscore. An arbitrary number of subsequent characters may be ASCII letters, underscores or numbers.

Legal identifiers: _test, A1234, _999

Illegal identifiers: 1ABC, XÖ, F$D, alpha.beta

14.3.5 Escape Sequences

\b	Backspace
\t	Tab
\n	New Line
\r	Carriage Return
\f	Form Feed
\"	Double quote
\'	Single quote
\nnn	Octal encoded character

14.3.6 String Literal

Quoted with single quotes, e.g. 'Text'

14.3.7 Decimal Literal

Supporting decimal syntax.

Legal decimal values: 1234.2345E+12, 1234.2345E12, 1234.2345e-12

14.3.8 Integral Number Literal

Supporting decimal, hexadecimal and octal syntax.

Legal decimal values: 0, 123

Legal octal values: 01, 00123

Legal hexadecimal values: 0x0dFa

14.3.9 Boolean Literal

Legal values: true, false

14.3.10 null Literal

Legal value: null

14.3.11 Qualified name

identifier(.identifier)*

Legal values: org.databene.benerator.Generator, Generator

14.3.12 Constructor invocation

Works like in Java, e.g.

```
new MyGenerator('alpha.txt', 23)
```

14.3.13 Bean construction

Instantiates an object by the class' default constructor and calling property setters, e.g.

```
new MyGenerator { filename='alpha.txt', limit=23 }
```

is executed like

```
MyGenerator generator = new MyGenerator();
generator.setFilename("alpha.txt");
generator.setLimit(23);
```

14.3.14 Method invocation

Can occur on static Java methods on a class or instance methods on an object.

Static method invocation: com.my.SpecialClass.getInstance()

instance method invocation: generator.generate()

14.3.15 Attribute access

Can occur on static Java attributes on a class or instance methods on an object.

Static attribute access: com.my.SpecialClass.instance

instance attribute access: user.name

14.3.16 Casts

Benerator can casts data types. Cast arguments are Benerator's simple types.

Examples: (date) '2009-11-23', (long) 2.34

14.3.17 Unary Operators

Arithmetic Negation: -

Bitwise complement: ~

Boolean Negation: !

14.3.18 Arithmetic Operators

Multiplication: *

Division: /

Modulo division: %

Addition: +, e.g. new Date() + 5000

Subtraction: -

14.3.19 Shift Operators

Left shift: <<

Right shift: >>

Right shift: >>>

14.3.20 Relation Operators

<=, <, >, =>

14.3.21 Equality Operators

==, !=

14.3.22 Bitwise Operators

And: &

Exclusive Or: ^

Inclusive Or: |

14.3.23 Boolean Operators

And: &&

Or: ||

DatabeneScript uses shortcut evaluation like C, C++ and Java: First it evaluates the left hand side of an operation. If the result is completely determined by the result, it does not evaluate the right hand side.

14.3.24 Conditional Expression

... ? ... : …, e.g. a>3 ? 0 : 1

14.3.25 Assignment

qualifiedName = expression

15 Command Line Tools

15.1 Benerator

benerator expects a descriptor file name as the only command line parameter, e.g. on Windows systems

```
benerator.bat test.ben.xml
```

or, on Unix and Mac OS X systems,

```
benerator.sh test.ben.xml
```

You can change default behavior by Java VM parameters, e.g.

```
benerator.bat -Dfile.encoding=iso-8859-1 -Djava.io.tmpdir="C:\temp" test.ben.xml
```

Validation can be turned off from the command line alternatively using a VM parameter:

```
mvn benerator:generate -Dbenerator.validate=false
```

or

```
benerator.sh myproject.ben.xml -Dbenerator.validate=false
```

15.2 DB Snapshot Tool

The DbSnaphotTool creates a snapshot of a full database schema and stores it in a DbUnit XML file. It is invoked from the command line in Windows by calling

```
snapshot.bat [VM-params] export-filename.dbunit.xml
```

or, on Unix and Mac OS X systems,

```
sh snapshot.sh [VM-params] export-filename.dbunit.xml
```

If the export filename is left out, the snapshot will be stored in a file called snapshot.dbunit.xml.

You need the following VM parameters to configure database access. Use them like -Ddb.user=me:

Parameter	Description
dbUrl	The JDBC URL of the database
dbDriver	The JDBC driver class name
dbUser	user name
dbPassword	user password
dbSchema	Name of the schema to extract (defaults to the user name)

15.3 XML Creator

The XMLCreator reads a XML Schema file and creates a number of XML files that comply to the schema. It can read XML annotations which provide benerator configuration in the XML schema file. It is invoked from

the command line and has the following parameter order:

```
createxml.bat <schemaUri> <root-element> <filename-pattern> <file-count>
[<properties file name(s)>]
```

Their meaning is as follows:

- schemaUri: the location (typically file name) of the XML schema file
- root-element: the XML element of the schema file that should be used as root of the generated XML file(s)
- filename-pattern: the naming pattern to use for the generated XML files. It has the form of a java.text.MessageFormat pattern and takes the number of the generated file as parameter {0}.
- file-count: the number of XML files to generate
- properties file name(s): an optional (space-separated) list of properties files to include in the generation process

Under Windows, an example call would be:

```
createxml.bat myschema.xsd product-list products-{0}.xml 10000
perftest.properties
```

or, on Unix and Mac OS X systems,

```
sh myschema.xsd product-list products-{0}.xml 10000 perftest.properties
```

for generation 10,000 XML files that comply to the XML Schema definition in file myschema.xsd and have product-list as root element. The files will be named products-1.xml, products-2.xml, products-3.xml, ...

16 Domains

benerator domains are a vehicle for defining, bundling and reusing domain specific data generation, e.g. for personal data, addresses, internet, banking, telecom. They may be localized to specific languages and be grouped to hierarchical datasets, e.g. for continents, countries and regions.

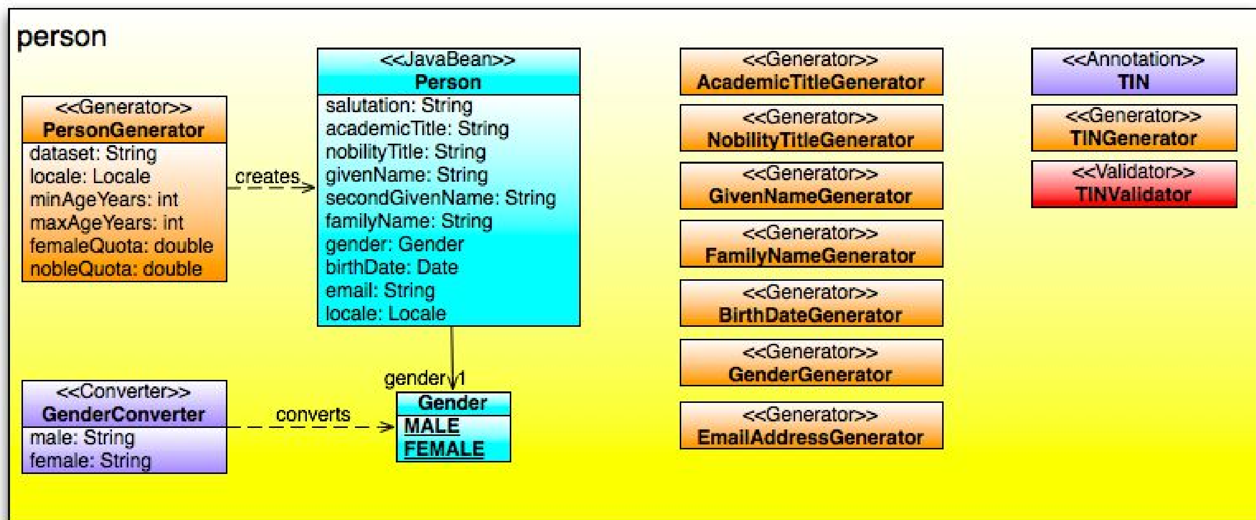
benerator comes packaged with several domains that have simple implementation of specific data generation and may serve as a base for deriving own specific data generation domains:

- **person**: Data related to a person
- **address**: Data related to contacting a person by post
- **net**: Internet and network related data
- **finance**: finance data
- **organization**: Organization data
- **product**: Product-related data

16.1 person domain

The person domain has three major components:

- PersonGenerator: Generates Person beans
- AcademicTitleGenerator: Generates academic titles
- NobilityTitleGenerator: Generates nobility titles
- GivenNameGenerator: Generates given names
- FamilyNameGenerator: Generates family names
- BirthDateGenerator: Generates birth dates
- GenderGenerator: Generates Gender values
- EmailAddressGenerator: Generates Email addresses
- GenderConverter: Converts Gender values to predefined strings
- TIN: Marks a Java attribute or property as a European Tax Identification Number
- TINGenerator: Creates European Tax Identification Numbers
- TINValidator: Validates European Tax Identification Numbers



16.1.1 PersonGenerator

Creates Person beans to be used for prototype-based data generation. It can be configured with dataset and locale property. The generated Person JavaBeans exhibits the properties salutation, title (both locale-dependend), givenName, familyName (both dataset-dependent), gender and birthDate. If the chosen dataset definition provides name weights, benerator generates person names according to their statistical probability. Of course, gender, salutation and givenName are consistent.

You can use the PersonGenerator like this:

```
<import domains="person"/>
<generate type="user" count="5" consumer="ConsoleExporter">
  <variable name="person" generator="PersonGenerator" dataset="FR" locale="fr"/>
```

```
  <attribute name="salutation" source="person.salutation" />
  <attribute name="name" script="{person.givenName +' '+ person.familyName}}" />
</generate>
```

to get output similar to this:

```
user[salutation=Mr, name=David Morel]
user[salutation=Mr, name=Robert Robert]
user[salutation=Mr, name=Eric Morel]
user[salutation=Mr, name=Patrick Lefebvre]
user[salutation=Mme, name=Helene Fournier]
```

16.1.2 PersonGenerator Properties

The PersonGenerator can be configured with several properties:

Property	Description	Default Value
dataset	Either a region name or the two-letter-ISO-code of a country, e.g. US for the USA. See Region Nesting.	The user's default country
locale	Two-letter-ISO-code of the language in which to create salutation and titles, e.g. en for English	The user's default language
minAgeYears	The minimum age of generated persons	15
maxAgeYears	The maximum age of generated persons	105
femaleQuota	The quota of generated women (1 → 100%)	0.5
nobleQuota	The quota of generated noble persons (1 → 100%)	0.005

16.1.3 Person Class

The Person class has the following properties:

property name	type	property description
salutation	String	Salutation (e.g. Mr/Mrs)
academicTitle	String	Academic title (e.g. Dr)
nobilityTitle	String	Nobility title (like Duke)
givenName	String	Given name ('first name' in western countries)
secondGivenName	String	An eventual second given name
familyName	String	Family name ('surname' in western countries)
gender	Gender	Gender (MALE or FEMALE)
birthDate	Date	Birth date
email	String	eMail address
locale	Locale	Language of the person instance (used e.g. for salutation)

16.1.4 Supported countries

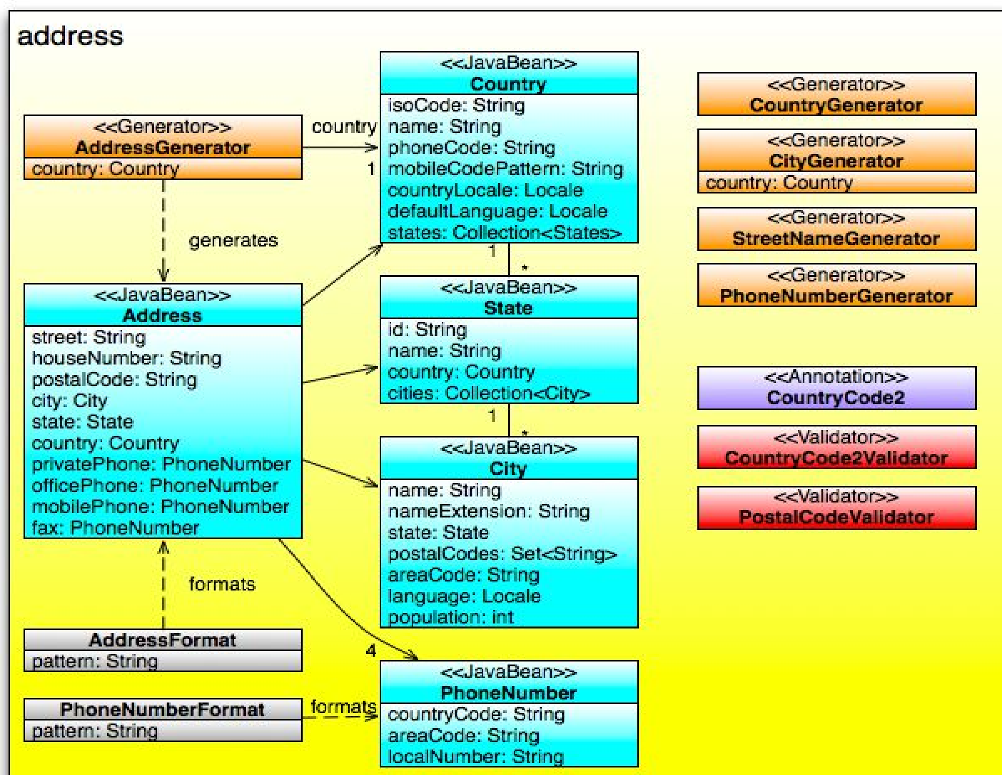
country	code	remarks

Austria	AT	most common 120 given names with absolute weight, most common 40 family names with absolute weight
Australia	AU	most common 40 given names (unweighted), most common 20 family names with absolute weight
Brazil	BR	most common 100 given names (unweighted), most common 29 family names (unweighted)
Canada	CA	most common 80 given names (unweighted), most common 20 family names (unweighted). No coupling between given name locale and family name locale
Switzerland	CH	most common 30 given names with absolute weight, most common 20 family names with absolute weight
Czech Republic	CZ	most common 20 given names with absolute weight, most common 20 family names with absolute weight. Female surnames are supported.
Spain	ES	most common 40 given names (unweighted), most common 40 family names with absolute weight
Finland	FI	most common 785 given names (unweighted), most common 448 family names (unweighted)
France	FR	most common 100 given names (unweighted), most common 25 family names with relative weight
Germany	DE	most common 1998 given names with absolute weight, most common 3421 family names with absolute weight2
Israel	IL	264 given names (unweighted), most common 30 family names with relative weight
India	IN	most common 155 given names (unweighted), most common 50 family names (unweighted)
Italy	IT	most common 60 given names (unweighted), most common 20 family names (unweighted)
Japan	JP	Kanji letters. Most common 109 given names (unweighted), most common 50 family names with absolute weight
Netherlands	NL	3228 given names (unweighted), most common 10 family names with absolute weight
Norway	NO	most common 300 given names (unweighted), most common 100 family names with absolute weight
Poland	PL	most common 67 given names with absolute weight, most common 20,000 family names with absolute weight. Female surnames are supported.
Russia	RU	Cyrillic letters. Most common 33 given names with relative weight, most common 20 family names with relative weight. Female surnames are supported.
Sweden	SE	779 given names (unweighted), most common 22 family names with relative weight
Turkey	TR	1077 given names (unweighted), 37 family names (unweighted)

United Kingdom	GB	most common 20 given (unweighted), most common 25 family names (unweighted)
USA	US	most common 600 given names and most common 1000 family names both with absolute weight

16.2 Address domain

- **AddressGenerator**: Generates addresses that match simple validity checks: The City exists, the ZIP code matches and the phone number area codes are right. The street names are random, so most addresses will not stand validation of real existence.
- **PhoneNumberGenerator**: Generates land line telephone numbers for a country
- **MobilePhoneNumberGenerator**: Generates mobile phone numbers for a country
- **CountryGenerator**: Generates countries
- **CountryCode2**: Annotation that marks a Java attribute or property as ISO-3166-1 alpha-2 code
- **CountryCode2Validator**: Java Bean Validation ConstraintValidator for ISO-3166-1 alpha-2 codes
- **CityGenerator**: Generates Cities for a given country
- **StreetNameGenerator**: Generates street names for a given country
- **PostalCodeValidator**: Validates if a given postal code is valid in a given country



The following countries are supported:

country	code	remarks
Germany	DE	Valid ZIP codes and area codes, no assurance that the street exists in this city or the local phone number has the appropriate length
USA	US	Valid ZIP codes and area codes, no assurance that the street exists in this city.
Brazil	BR	Valid ZIP codes and area codes, no assurance that the street exists in this city or the local phone number has the appropriate length

16.3 net domain

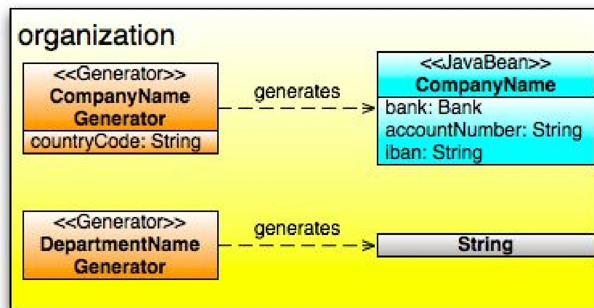
The net domain provides the

- **DomainGenerator**, which generates Internet domain names

16.4 organization domain

Provides the following generators:

- **CompanyNameGenerator**, a generator for company names.
- **DepartmentNameGenerator**, a generator for department names



If you use the CompanyNameGenerator like this:

```
<import domains="organization" />
<generate type="company" count="5" consumer="ConsoleExporter">
  <attribute name="name" generator="CompanyNameGenerator"
      dataset="DE" locale="de_DE"/>
</generate>
```

you get output like this:

```
company[name=Belanda Aktiengesellschaft & Co. KG]
company[name=MyWare Technologies GmbH]
company[name=WebBox AG]
company[name=Altis AG]
company[name=Ernst Fischer Technik GmbH]
```

Company names can be generated for the following countries:

country	code	remarks
Germany	DE	none
USA	US	none

The Generator creates objects of type CompanyName, consisting of **shortName**, **sector**, **location** and **legalForm** information. You can make use of the object as a whole which is converted to a string automatically using the '**fullName**' version as shown in the example above. But you can also make direct use of the basic properties:

```
<import domains="organization" />
```

```
<generate type="company" count="5" consumer="ConsoleExporter">

  <variable name="c" generator="CompanyNameGenerator"
      dataset="DE" locale="de_DE"/>

  <attribute name="name" script="c.shortName + ' ' + c.legalForm" />

</generate>
```

16.5 finance domain

Generates and validates finance related data:

The following classes are provided:

- **BankAccountGenerator**: Generates BankAccount JavaBeans
- **BankAccountValidator**: Validates BankAccount JavaBeans
- **CreditCardNumberGenerator**: Generates strings which represent credit card numbers
- **CreditCardNumberValidator**: Validates strings as credit card numbers
- **IBAN**: Annotation for Java Bean Validation, marking a Java attribute/property as IBAN
- **IBANValidator**: Validates strings with IBANs

16.6 product domain

The product package provides you with Generator classes for EAN codes:

- **EAN8Generator**: Generates 8-digit EAN codes
- **EAN8**: Annotation that marks a Java attribute or property as 8-digit-EAN for bean validation
- **EAN8Validator**: Validates 8-digit EAN codes
- **EAN13Generator**: Generates 13-digit EAN codes
- **EAN13**: Annotation that marks a Java attribute or property as 13-digit-EAN for bean validation
- **EAN13Validator**: Validates 13-difit EAN codes
- **EANGenerator**: Generates both 8-digit and 13-digit EAN codes
- **EAN**: Annotation that marks a Java attribute or property as an EAN for bean validation
- **EANValidator**: Validates 8- and 13-digit-EAN codes

Each generator has a property **'unique'**: If set to true the generator assures that no two identical EAN codes are generated.

16.7 br domain

Provides classes specific to Brazil:

- **CPNJ**: Annotation to mark a Java attribute or property as a CPNJ (Cadastro Nacional da Pessoa Jurídica)
- **CPNJGenerator**: Generates CPNJs
- **CPNJValidator**: Validates CPNJs and can be used as Databene validator and as ConstraintValidator in Java Bean Validation (JSR 303)
- **CPF**: Annotation to mark a Java attribute or property as a CPF (Cadastro de Pessoa Fisica)
- **CPFGenerator**: Generates CPFs
- **CPFValidator**: Validates CPFs

16.8 us domain

Provides classes specific for the United States of America:

- **SSN**: Annotation to mark a Java attribute or property as a Social Security Number
- **SSNGenerator**: Generates Social Security Numbers
- **SSNValidator**: Validates Social Security Numbers and can be used as Databene validator and as ConstraintValidator in Java Bean Validation (JSR 303)

17 Component Reference

benerator has lots of predefined generators which are available implicitly from the descriptor. Most of them only need to be created explicitly when using the benerator API programmatically.

17.1 Generators

17.1.1 Domain Generators

For domain-specific generators (e.g. person, address, finance), see **'Domains'**.

17.1.2 Common Id Generators

Benerator contains the following common predefined and platform-independent generators:

- **IncrementalIdGenerator**: Creates consecutive id values, starting with 1 by default.
- **UUIDGenerator**: Creates UUIDs by the JDK class java.util.UUID
- **HibUUIDGenerator**: Creates UUIDs like the Hibernate UUID key generator
- **LocalSequenceGenerator**: Mimics the behavior of a (named) database sequence on a single client VM. Its property 'cached' (true by default) specifies if sequence value changes shall be persisted immediately or in the end.

17.1.3 Database-related Generators

See 'Using databases'

17.1.4 simple type generators

- **CharacterGenerator**:
- **IncrementGenerator**: Generates numbers starting with one and incrementing the number on each subsequent call
- **StringGenerator**: Generates strings based on character set, prefix, suffix and length characteristics. This is the typical component for generating code numbers. Properties: charSet (regular expression for a character class), locale, unique, ordered, prefix, minInitial, suffix, minLength, maxLength, lengthGranularity, lengthDistribution
- **RegexStringGenerator**: Generates strings that match a given regular expression. This is the typical component for generating strings that are composed of different sub patterns. Properties: pattern (regular expression), unique, ordered, locale, minLength, maxLength
- **MessageGenerator**: Composes strings using a MessageFormat
- **LuhnGenerator**: Generates Luhn-valid strings like credit card numbers

17.1.5 current date / time generators

- **CurrentDateGenerator**: Generates java.util.Date objects that represent the current date
- **CurrentDateTimeGenerator**: Generates java.util.Date objects that represent the current date and time
- **CurrentMilliTimeGenerator**: Generates long values that denote the number of milliseconds since 1970-01-01 00:00:00
- **CurrentNanoTimeGenerator**: Generates long values that denote a number of milliseconds since an arbitrary point in time (possible even in the future, so values may be negative)

- **CurrentTimeGenerator**: Generates java.util.Date objects that represent the current time of the day

17.1.6 arbitrary date / time generators

- **DateGenerator**: Generates date values that represent a certain time at a certain day based on a common Distribution
- **DayGenerator**: Generates date values that represent „day“ dates – dates at midnight
- **DateTimeGenerator**: Generates date values with date and time configurable independently
Its properties are: minDate, maxDate, dateGranularity, dateDistribution, minTime, maxTime, timeGranularity, timeDistribution. For a 9-to-5 datetime on odd days in August 2010, configure

```
<bean id="dtGen" class="DateTimeGenerator">
      <property name='minDate'          value='2010-08-01'/>
      <property name='maxDate'          value='2010-08-31'/>
      <property name='dategranularity'  value='00-00-02'  />
      <property name='dateDistribution' value='random'    />
      <property name='minTime'          value='08:00:00'  />
      <property name='maxTime'          value='17:00:00'  />
      <property name='timeGranularity'  value='00:00:01'  />
      <property name='timeDistribution' value='random'    />
</bean>
```

17.1.7 file related generators

- **FileGenerator**: generates java.io.File objects representing files in a given directory structure
- **FileNameGenerator**: generates file names representing files in a given directory structure
- **TextFileContentGenerator**: provides text file contents as String
- **BinaryFileContentGenerator**: provides binary file contents as byte[]

17.1.8 State Generators

- **StateGenerator**: Generates states based on a state machine
- **StateTransitionGenerator**: Like the StateGenerator, but generating Transition objects

17.1.9 Seed Based Generators

- **SeedWordGenerator**: Generates new word based on rules derived from a dictionary.
- **SeedSentenceGenerator**: Generates sentences based on rules derived from a text file.

17.2 Distributions

A Distribution describes stochastic properties for distributing the data that benerator generates. You can use the predefined distributions or implement and introduce custom implementations. The most important types of distribution are *Sequence*, *WeightFunction* and *CumulativeDistributionFunction*.

A Distribution implements a common concept for generating numbers or taking values from a data source and providing them in a rearragned order or distribution with similar semantics as the number generation feature.

As an example, a 'Skip2' sequence might generate numbers with an increment of 2: 1, 3, 5, 7,… When it is used to redistribute given data item1, item2, item3, item4,... , it would provide the values item1, item3, ...

While most Distribution components implement number generation as well data rearrangement, they are not required to support both concepts.

All Distributions listed below are included in the default imports.

17.2.1 Memory consumption

Distributions that are based on number generation may adopt data redistribution by simply loading all available data into a long list in RAM and then using their number generation feature to determine indices of the data to provide. If the data amount is large, you may get memory problems. In order to provide an easy start, Benerator reduces the default size of these lists to 100,000 elements, prints out an error message if the number is exceeded, but simply continues to work with the reduced amount of data. You can allow Benerator to use a larger cache by adding a benerator.cacheSize to your BENERATOR_OPTS, e.g. -Dbenerator.cacheSize=2000000. If this makes you run into an OutOfMemoryError, check the 'Troubleshooting' section on how to allocate a larger Java heap in Benerator.

17.2.2 Sequences

Sequences reflect the idea of a mathematical sequence. The primary focus in number generation, but they can be applied for data redestribution as well. Most sequences have a default instance which can be used by their literal, e.g. distribution="random" uses the 'random' literal for the Distribution defined in the class RandomSequence.

Class	RandomSequence
Description	Creates uniformly distributed random values
Default Instance	random

Class	CumulatedSequence
Description	Creates random values with a bell-shape probability distribution
Default Instance	cumulated

Class	StepSequence
Description	Depending on the settings of property 'delta', it starts with the min or max value of the specified range. With each further invocation, the 'increment' value is added. If addition makes the current value exceed the specified number range, the Sequence becomes unavailable. So the numbers provided are unique. Example: increment = -2, range=1..7: 7, 5, 3, 1
Default Instances	increment: Uses delta = 1 to create incremental values, e.g. 1, 2, 3,... **step**: Old name of the 'increment' instance (provided only for backwards compatibility)

Property	Property Description	Default Value
delta	The difference between the next value and the previous one	1

Class	**RandomWalkSequence**	
Description	Starting with an → **initial** value, a random value between → **minStep** and → **maxStep** is added on each subsequent invocation	
Default Instance	**randomWalk**	
Property	**Property Description**	**Default Value**
minStep	The maximum delta between the next and the previous value	-1
maxStep	The maximum delta between the next and the previous value	1
initial	If no initial value was configured explicitly, number generation starts with the min, max or medium value of the specified range – depending on the settings of minStep and maxStep	null

Class	**ShuffleSequence**	
Description	Can be used to iterate quickly through a large number range with avoiding duplicate values. It starts from an offset of 0 and iterates the number range with a fix increment. After the range is covered, it increases the offset by one and reiterates the range. When the offset reaches the same value as the increment, it is set back to 0 again. For an increment of 3 in a range 1..7, the generated numbers would be 1, 4, 7, 2, 5, 3, 6, 1, 4, ...	
Default Instance	**shuffle**	
Property	**Property Description**	**Default Value**
increment	See the class description	2

Class	**WedgeSequence**
Description	Starting with first the lowest, then the highest available number, this alternatively provides increasing small numbers and decreasing large numbers until they converge in the middle and the Sequence becomes unavailable. So generation is unique. For a number range 1..7, the generated numbers would be: 1, 7, 2, 6, 3, 5, 4.
Default Instance	**wedge**

Class	**BitReverseSequence**
Description	Creates numbers by continually increasing an internal counter and providing its value in bit-reversed order. This stops when each available number has been generated once, thus providing unique number generation. This comes close to a unique random distribution.
Default Instance	**bitreverse**

Class	**ExpandSequence**
Description	Distributes numbers or data of unlimited volume in a unique or non-unique manner, by starting with a limited lower range and continuously expanding data region as data is

	generated. This comes close to a unique random distribution and can be used to iterate over very huge amounts of data.	
Default Instance	**expand**	
Property	**Property Description**	**Default Value**
cacheSize	The maximum number of elements to keep in RAM at the same time	100
bucketSize	The size of 'buckets' over which to distribute the iterated data. The smaller the number, the more buckets are used and the more erratic the generated data looks.	10
duplicationQuota	The probability by which a data element will be reused in a later call	0

Class	**HeadSequence**	
Description	When applied to a data source or generator, only the first few elements are provided. The number of elements is defined by the **size** property.	
Default Instance	**head**	
Property	**Property Description**	**Default Value**
size	The size of the buffer	1

Class	**LiteralSequence**	
Description	Defines a number sequence using a comma-separated list literal. Example: use distribution="new LiteralSequence('2,3,5,7,11')" for generating the numbers 2, 3, 5, 7, 11 consecutively.	
Default Instance	**–**	
Property	**Property Description**	**Default Value**
spec	A comma-separated list with all values in the order in which they shall be provided ,e.g. '2,3,5,7,11'	–

Class	**WeightedNumbers**	
Description	Creates numbers based on a weighted-number literal , e.g. '1^70, 3^30' for generating 70% '1' values and 30% '3' values. This is a convenient and simple approach for controling parent-child cardinalities in nested data generation. Short form, e.g.: distribution="new WeightedNumbers('1^70,3^30')"	
Default Instance	**–**	
Property	**Property Description**	**Default Value**
spec	A weighted-number literal. It lists weighted values in a comma-separated list. Each weighted value is specified by the numeric value followed by a circumflex (^) and the weight value, for example '1^70,3^30'	–

Class	**FibonacciSequence**
Description	Generates numbers based on the Fibonacci Sequence
Default Instance	fibonacci:

Class	PadovanSequence
Description	Generates numbers based on the Padovan Sequence
Default Instance	padovan

17.2.3 CumulativeDistributionFunction

The CumulativeDistributionFunction is another special case of a Distribution, which allows for continuous value generation as opposed to Sequence and WeightFunction.

17.2.4 ExponentialDensityIntegral

Inverse of the integral of the probability density f(x) = a e^{-ax} (x > 0), which resolves to F^{-1}(x) = - log(1 - x) / a.

17.2.5 Weight Functions

Weight funtions are another special case of Distributions. They are based on a function which is supposed to allow contibuous value generation, but since Benerator needs to perform a numerical integration for deriving random values, a granularity must be applied. This way, the generated value set is quantized. Another drawback of the approach is that fine-grained generation is memory-consuming and slow.

Thus, it is recommended to avoid weight functions if possible and choose a similar Sequence or CumulativeDistributionFunction instead.

17.2.6 GaussianFunction

This implements the well-known Gaussian Function.

Full class name: org.databene.benerator.distribution.function.GaussianFunction

Parameters: average [, deviation]

Example:

```
<import class="org.databene.benerator.distribution.function.*"/>
...
<attribute name="price" type="big_decimal"
    min="0.1" max="99.90" granularity="0.1"
    distribution="new GaussianFunction(50,20)"/>
```

17.2.7 ExponentialFunction

The Exponential Function.

Full class name: org.databene.benerator.distribution.function.ExponentialFunction

Parameters: [scale,] frequency

Example:

```
<import class="org.databene.benerator.distribution.function.*"/>
...
<attribute name="category" type="char" values="A,B,C"
    distribution="new ExponentialFunction(0.5)"/>
```

17.2.8 DiscreteFunction

Discrete Function that specifies an explicit weight for each possible value

Full class name: org.databene.benerator.distribution.function.DiscreteFunction

Parameters: weight1 [, weight2 [, weight3 ...]]

Example:

```
<import class="org.databene.benerator.distribution.function.*"/>

...

<attribute name="rating" type="int" min="1", max="3"
    distribution="new DiscreteFunction(1, 2, 1)"/>
```

17.3 Converters

Benerator supports two different types of converter interfaces:

- org.databene.commons.Converter
- java.text.Format

17.3.1 Databene Converters

The following converter classes are located in the package **org.databene.commons.converters** and are imported with the default imports:

- **ByteArrayToBase64Converter**: Converts byte arrays to strings which are base-64-encoded
- **ToLowerCaseConverter**: Converts strings to lowercase
- **ToUpperCaseConverter**: Converts strings to uppercase
- **LiteralParser**: Parses strings as numbers, strings, dates and times
- **MessageConverter**: Converts an object, wrapping it with a message string, using a java.text.MessageFormat
- **PropertyResourceBundleConverter**: Uses a Java PropertyResourceBundle to translate keywords to translations in a given Java Locale
- **ToStringConverter**: Converts arbitrary objects to strings
- **UniqueStringConverter**: Assures uniqueness for all processed Strings by appending unique numbers to recurring instances (attention: limited to a few 100.000 elements)
- **URLEncodeConverter**: Applies a URL encoding to strings
- **URLDecodeConverter**: decodes URL encoded strings
- **PrintfConverter**: formats objects using a pattern in printf format
- **RegexReplacer**: Uses a regular expression to replace parts of the processed strings
- **SubstringExtractor**: Extracts substrings from strings. It has the properties '**from**' and '**to**'. If '**to**' is not set, it extracts from '**from**' until the end. If '**to**' or '**from**' is negative, it denotes a backwards position count, making e.g. -1 the last character position.
- **EscapingConverter**: Escapes strings in Java style, like "A\tB"
- **Number2CharConverter**: Converts a number to a character of the corresponding ASCII code
- **Char2StringConverter**: Converts a character to a string of length 1
- **EscapingConverter**: Escapes control codes in a string in C and Java style, e.g. with \r, \n, \t
- **Number2CharConverter**: Converts a number to a character with the corresponding ASCII code, e.g. 65 → 'A'

The package **org.databene.text** provides the following converters:

- **DelocalizingConverter**: Converts strings with non-ASCII letters to ASCII strings, e.g. Müller → Mueller, Sœr → Soer
- **NameNormalizer**: Normalizes a string by trimming it, normalizing inner white space and formatting each word to start with an uppercase character and continue with lowercase characters
- **NormalizeSpaceConverter**: Trims a string and normalizes inner white space to one space character
- **ToHexConverter**: Renders characters, strings snd integral numbers in hexadecimal representation

In the package **org.databene.benerator.primitive.number** there are two converters that can be used to quantize numerical values:

- **FloatingPointQuantizer**, **IntegralQuantizer, NumberQuantizer**: Quantize numbers to be a **min** value plus an integral multiple of a **granularity**
- **NoiseInducer**: Adds numerical noise to numbers. The noise characteristics can be configured with the properties minNoise, maxNoise, noiseGranularity and noiseDistribution. When setting the boolean property relative to true, noise is relative, where maxCount=1 corresponds to 100% noise-to-signal ratio. If relative=false, the absolute value of the noise is added or subtracted. Example:

NoiseInducer example:

```
<bean id="inducer" class="org.databene.benerator.primitive.number.NoiseInducer">
    <property name="minNoise" value="-0.2"/>
    <property name="maxNoise" value="0.2"/>
    <property name="noiseGranularity" value="0.01"/>
    <property name="noiseDistribution" value="cumulated"/>
    <property name="relative" value="true"/>
</bean>

<generate count="5" consumer="ConsoleExporter">
    <attribute name="x" type="int" constant="100" converter="inducer" />
</generate>
```

produces the result:

```
entity[x=99]
entity[x=105]
entity[x=92]
entity[x=104]
entity[x=99]
```

17.3.2 Java Formats

Beware that the java.text.Format classes are not thread-safe!

- **SimpleDateFormat**: Uses a pattern to format dates as strings
- **DecimalFormat**: Uses a pattern to format numbers as strings

17.4 Validators

17.4.1 Domain Validators

For the validators from the domains see 'Domains'

17.4.2 Common validators

- **CharacterRangeValidator**: Validates if a character is in a certain range
- **NotNullValidator**: Requires the validated data to be not null
- **StringLengthValidator**: Limits allowed strings to a minimum and/or maximum length
- **StringValidator**: Validates string by min length, max length and a charactor validator
- **UniqueValidator**: Requires data to be unique (attention: limited to some 100.000 elements)
- **UnluckyNumberValidator**: Checks if a String contains an 'unlucky' number like 13 in western cultures or 4 in east-asian cultures
- **DayOfWeekValidator**: Accepts only Dates of certain (configurable) weekdays
- **RegexValidator**: Validates if a string matches a regular expression
- **LuhnValidator**: Checks if a number string (e.g. credit card number) is Luhn-valid

17.4.3 Tasks

- **FileJoiner**: Joins several files (**sources**) into a **destination** file, optionally **append**ing the joint data to an existing destination file, or overwriting it. If **deleteSources** is set to true, the sources are deleted afterwards.
- **FileDeleter**: Deletes a number of **files**.

17.5 Consumers

A Consumer consumes generated data and usually is used for exporting or persisting the data.

17.5.1 LoggingConsumer

Class Name	LoggingConsumer
Import	default
Class Description	Logs all Consumer invocations to a logger

17.5.2 ConsoleExporter

Class Name	ConsoleExporter	
Import	default	
Class Description	Prints entities in the console	
Constructors	Default constructor Constructor with 'limit' argument (see below)	
Property	**Property Description**	**Default Value**
limit	The maximum number of entries per type to print out	unlimited
nullString	Text to represent *null* values	""
datePattern	The pattern to render date values	"yyyy-MM-dd"
timePattern	The pattern to render time values	"HH:mm:ss"
timestampPattern	The pattern to render timestamp values	"yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS"
decimalPattern	The pattern to render decimal values	System default
decimalSeparator	The decimal separator to use for decimal values	System default
integralPattern	The pattern to integral number values	System default

17.5.3 JavaInvoker

Class Name	DbUnitEntityExporter	
Import	<import platforms="java"/>	
Class Description	Maps entity components to method parameters and invokes a method on a Java object with these parameters.	
Property	**Property Description**	**Default Value**
target	The Java object on which to invoke the method	
methodName	The name of the Java method to invoke	

Usage example:

```
<bean id="service" spec="..." />
<bean id="invoker" spec="new JavaInvoker(ejb, 'enrolCustomer')" />
```

17.5.4 DbUnitEntityExporter

Class Name	DbUnitEntityExporter	
Import	<import platforms="dbunit"/>	
Class Description	Exports entities to a file in DbUnit XML format.	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"data.dbunit.xml"
encoding	The character encoding to use for the file	The system default

17.5.5 XMLEntityExporter

Class Name	XMLEntityExporter	
Import	<import platforms="xml"/>	
Class Description	Exports entities to an XML file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"export.xml"
encoding	The character encoding to use for the file	The system default

17.5.6 NoConsumer

Class Name	NoConsumer
Import	default
Class Description	In some cases a pseudo <generate> statements acts as a mechanism to perform a loop. In such cases a consumer does not make sense but causes Benerator to emit a warning „No consumers defined for <loop name>“. In order to avoid this warning, you can use the NoConsumer class, which is an empty implementation of the Consumer interface.

17.5.7 ScriptedEntityExporter

Class Name	ScriptedEntityExporter	
Import	<import platforms="script"/>	
Class Description	Exports entities to a file in custom format, rendered using a script language, e.g. FreeMarker. Three different script expressions may be applied for header (headerScript property), entity (partScript) and footer (footerScript).	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	
encoding	The character encoding to use for the file	The system default
headerScript	Script to format an eventual header line	
partScript	Script to format an exported entity	
footerScript	Script to format an eventual footer line	
nullString	Text to represent *null* values	""
datePattern	The pattern to render date values	"yyyy-MM-dd"
dateCapitalization	The capitalization to use when rendering a month name in a date: 'upper', 'lower' or 'mixed'	mixed
timePattern	The pattern to render time values	"HH:mm:ss"
timestampPattern	The pattern to render timestamp values	"yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS"
timestamp Capitalization	The capitalization to use when rendering a month name in a timestamp: 'upper', 'lower' or 'mixed'	mixed
decimalPattern	The pattern to render decimal values	System default
decimalSeparator	The decimal separator to use for decimal values	System default
integralPattern	The pattern to integral number values	System default

17.5.8 FixedWidthEntityExporter

Class Name	FixedWidthEntityExporter	
Import	<import platforms="fixedwidth"/>	
Class Description	Exports entities to a fixed column width file.	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"export.fcw"
columns	A comma-separated list of column format specifications	
encoding	The character encoding to use for the file	System default
lineSeparator	The line separator to use in the generated file	System default
append	If set to true, data is appended to existing files, otherwise existing files are overwritten	false
nullString	Text to represent *null* values	""
datePattern	The pattern to render date values	"yyyy-MM-dd"
dateCapitalization	The capitalization to use when rendering a month name in a date: 'upper', 'lower' or 'mixed'	mixed
timePattern	The pattern to render time values	"HH:mm:ss"
timestampPattern	The pattern to render timestamp values	"yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS"
timestamp Capitalization	The capitalization to use when rendering a month name in a timestamp: 'upper', 'lower' or 'mixed'	mixed
decimalPattern	The pattern to render decimal values	System default
decimalSeparator	The decimal separator to use for decimal values	System default
integralPattern	The pattern to integral number values	System default

The line format is described as a comma-separated list of property names with format spec, e.g. name[20],age[3r],points[5.2r0]. The format spec consists of

- [] brackets
- the (required) column width
- an optional alignment flag l, r or c (for left, right, center), left by default
- an optional pad character, space by default

So a property configuration of name[20],age[3r],points[5.2r0] would resolve to three columns,

first, a name entry, padded to 20 columns using spaces (default), aligned to the left (default)

second, an age entry, padded to 3 columns using spaces (default), aligned to the right

third, a points column, padded to 5 columns using zeros, having two fraction digits, aligned to the right

and would be rendered like this:

```
Alice Hamilton       2310.05
Bob Durand           4601.23
```

```
Helmut Schmidt      10226.14
```

17.5.9 XLSEntityExporter

Class Name	XLSEntityExporter	
Import	<import platforms="xls"/>	
Class Description	Exports entities to Excel XLS files. For using this exporter you need to add the Apache POI library to the benerator's lib directory.	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"export.xls"
columns	A comma-separated list of column names	
nullString	Text to represent *null* values	""

17.5.10 CSVEntityExporter

Class Name	CSVEntityExporter	
Import	<import platforms="csv"/>	
Class Description	Exports entities to a CSV file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"export.csv"
columns	A comma-separated list of column names	
headless	Flag to leave out column headers	false
separator	The character to use as column separator	","
encoding	The character encoding to use for the file	System default
lineSeparator	The line separator to use in the generated file	System default
endWithNewLine	Specifies if the last row in the file should end with a line break	false
append	If set to true, data is appended to existing files, otherwise existing files are overwritten	false
nullString	Text to represent *null* values	Empty string
quoteEmpty	When set to 'true', empty strings are formatted with double quotes ("",""), otherwise an empty field (,)	false
datePattern	The pattern to render date values	"yyyy-MM-dd"
dateCapitalization	The capitalization to use when rendering a month name in a date: 'upper', 'lower' or 'mixed'	mixed
timePattern	The pattern to render time values	"HH:mm:ss"
timestampPattern	The pattern to render timestamp values	"yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS"
timestamp Capitalization	The capitalization to use when rendering a month name in a timestamp: 'upper', 'lower' or 'mixed'	mixed
decimalPattern	The pattern to render decimal values	System default
decimalSeparator	The decimal separator to use for decimal values	System default
integralPattern	The pattern to integral number values	System default

17.5.11 SQLEntityExporter

Class Name	SQLEntityExporter	
Import	<import platforms="db"/>	
Class Description	Exports entities as 'INSERT' commands to a SQL file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to create	"export.sql"
encoding	The character encoding to use for the file	System default
lineSeparator	The line separator to use in the generated file	System default
append	If set to true, data is appended to existing files, otherwise existing files are overwritten	false
dialect	The SQL dialect to use in the generated file. Available values: db2, derby, firebird, hsql, h2, oracle, postgres, sql_server	
datePattern	The pattern to render date values	"yyyy-MM-dd"
timePattern	The pattern to render time values	"HH:mm:ss"
timestampPattern	The pattern to render timestamp values	"yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS"
decimalPattern	The pattern to render decimal values	System default
decimalSeparator	The decimal separator to use for decimal values	System default
integralPattern	The pattern to integral number values	System default

17.6 EntitySources (Importers)

benerator provides the following implementations of the EntitySource interface:

17.6.1 DbUnitEntitySource

Class Name	DbUnitEntitySource	
Import	<import platforms="dbunit"/>	
Class Description	Imports entities from a DbUnit XML file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to read	"export.sql"

17.6.2 CSVEntitySource

Class Name	CSVEntitySource	
Import	<import platforms="csv"/>	
Class Description	Imports entities from a CSV file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to read	"export.sql"
encoding	The character encoding used in the file	System default
separator	The line separator used in the file	","
columns	When set, the input file is expected to have no header row	

17.6.3 FixedColumnWidthEntitySource

Class Name	CSVEntitySource	
Import	<import platforms="fixedwidth"/>	
Class Description	Imports entities from a fixed column width file	
Property	**Property Description**	**Default Value**
uri	The URI of the file to read	"export.sql"
encoding	The character encoding of the file	System default
columns	The columns specification (See the FixedWidthEntityExporter for documentation)	

17.6.4 XLSEntitySource

Class Name	XLSEntitySource	
Import	<import platforms="xls"/>	
Class Description	Imports entities from an Excel(TM) document	
Property	**Property Description**	**Default Value**
uri	The URI of the file to read	

17.7 Benerator Utility Classes

17.7.1 RandomUtil

Class Name	RandomUtil
Import	<import class="org.databene.benerator.util.RandomUtil"/>
Class Description	Provides basic random functions
Method	**Method Description**
randomLong(min, max)	Returns a random long between min (inclusively) and max (inclusively)
randomInt(min, max)	Returns a random int between min (inclusively) and max (inclusively)
randomElement(value1, value2, ...)	Returns a random element of the ones listed as parameters
randomElement(List values)	Returns a random element from the 'values' list
randomIndex(List values)	Returns a random index for the 'values' list
randomDigit(min)	Returns a random numerical character with a value of at least min. Example: randomDigit(1) produces characters between '1' and '9'.
randomProbability()	Returns a float between 0 and 1
randomDate(min, max)	Returns a random date between min (inclusively) and max (inclusively)
randomFromWeightLiteral(literal)	Evaluates the weight literal and returns one of the specified values with the specified probability. Example literal: 'A'^3,'B'^1 will produce 75% of 'A' values and 25% of 'B' values.

17.8 Databene Commons Library

The library Databene commons is a general-purpose utility collection which also provides some features useful for data generation and manipulation. Its converters and validators are liste above, but there are some general utility classes too. They can be invoked directly using DatabeneScript or other supported script languages.

17.8.1 TimeUtil

Class Name	TimeUtil
Import	<import class="org.databene.commons.TimeUtil"/>
Class Description	Provides time and date related utility methods
Method	**Method Description**
currentYear()	returns the current year as integer value
today()	returns the current day as date object
yesterday()	returns the previous date as date object
tomorrow()	returns the next day as date object
date(year, month, day)	creates a date object for the specified day in the user's default time zone. 'month' is a zero-based integer, January = 0, February = 1, ...
gmtDate(year, month, day)	creates a date object for the specified day in the time zone GMT. 'month' is a zero-based integer, January = 0, February = 1, ...
date(year, month, day, hours, minutes, seconds, milliseconds)	creates a date object for the specified day and time in the user's default time zone. 'month' is a zero-based integer, January = 0, February = 1, ...
date(millis)	creates a date object in the user's default time zone. The time is specified as milliseconds since 1970-01-01
year(date)	returns the year of the specified date as integer
month(date)	returns the month of the specified date as integer
dayOfMonth(date)	returns the day of month of the specified date as integer
firstDayOfMonth(date)	returns the first day of the specified date's month
lastDayOfMonth(date)	returns the last day of the specified date's month
millis(year, month, day, hour, minute, second)	Calculates the number of milliseconds since 1970-01-01. 'month' is a zero-based integer, January = 0, February = 1, ...
time(hour, minute, second)	Creates a time object for the specified time
time(hour, minute, second, millisecond)	Creates a time object for the specified time
timestamp(year, month, day, hour, minute, second, nanosecond)	Creates a timestamp value for the specified time. 'month' is a zero-based integer, January = 0, February = 1, ...
currentTime()	Creates a time object representing the current time
midnightOf(date)	Rounds down a date value that may include a time to a value that represents midnight (time = 0)
addDays(date, noOfDays)	Calculates a date a given number of days past a given date
addMonths(date, noOfMonths)	Calculates a date a given number of months past a given date
addYears(date, noOfYears)	Calculates a date a given number of years past a given date

17.8.2 Period

Class Name	Period
Import	<import class="org.databene.commons.Period"/>
Class Description	Provides constants for some time periods
Invocation	**Description**
Period.SECOND.millis	The number of milliseconds in a second
Period.MINUTE.millis	The number of milliseconds in a minute
Period.HOUR.millis	The number of milliseconds in an hour
Period.DAY.millis	The number of milliseconds in a day
Period.WEEK.millis	The number of milliseconds in a week

17.8.3 IOUtil

Class Name	IOUtil
Import	<import class="org.databene.commons.IOUtil"/>
Class Description	Provides I/O related utility methods
Method	**Method Description**
isURIAvailable(uri)	Tells if the file specified by the given URI exists
getContentOfURI(uri)	Provides the content of the specified file as string
getBinaryContentOfUri(uri)	Provides the content of the specified file as byte array
getParentUri(uri)	Determines the parent URI (folder) of the specified URI
getProtocol(uri)	Determines the protocol specified in the URI
download(sourceUrl, targetFile)	Downloads the content of a remote URI to the local file system
copyFile(sourceUri, targetUri)	Copies a file on the local file system

17.8.4 CharUtil

Class Name	CharUtil
Import	<import class="org.databene.commons.CharUtil"/>
Class Description	Provides character related utility methods
Method	**Method Description**
ordinal(character)	Returns a character's ordinal as integer
character(ordinal)	Returns the character that corresponds to the given ordinal

18 Using DB Sanity

You can call Databene DB Sanity for checking preconditions before starting data generation and for verifying properness of the generated data in the end. For detailed information about DB Sanity, check its project homepage: http://databene.org/dbsanity. To use its functionality in Benerator, download and install dbsanity4ben from http://databene.org/dbsanity4ben.

For calling DB Sanity in a Benerator descriptor file, you define the checks and put them into a single XML file or distribute them over several ones and put them into a sub directory of your Benerator project, typically called 'dbsanity'.

Import the plugin functionality to your Benerator project using

```
<import platform="dbsanity" … />
```

Then you can call DB Sanity providing a reference to a database you have declared before:

```
<database id="db" … />
<dbsanity database="db" />
```

Alternatively, you can specify the environment and use a new database connection to perform data verification:

```
<dbsanity environment="mytestdb" />
```

You have the following configuration options:

Option	Description	Default Value
environment	The environment name with the configuration of the database to verify (see the 'database' chapter about environment definition).	-
database	The database to verify	-
in	The directory from which to read the	dbsanity
out	The directory in which to put the report	dbsanity-report
appVersion	An application version for which to perform the checks	*
tables	Comma-separated list of tables on which to restrict the checks	-
tags	Comma-separated list of tags on which to restrict the checks	-
skin	The DB sanity skin to use for reports	online
locale	The locale in which to render values	default locale
mode	DB Sanity's execution mode	default
onError	Configures how to react to a requirements violation	See the chapter 'Error Handling'

19 Maven Benerator Plugin

The benerator plugin enables you to attach benerator to your build cycle or simply use maven and its configuration capabilities for benerator setup and invocation.

You can use the plugin to

- Invoke Benerator with a descriptor file.
- Create XML files from an XML Schema file, supporting XML Schema annotations for generation setup.
- Create database snapshots in Excel, SQL or DbUnit data file format.

19.1 System Requirements

Maven 2.0.8 or newer

JDK 1.6 or newer

19.2 Getting started

What ever goal you want to accomplish with Maven Benerator Plugin, you need to create a Maven project first. If you are about to create a completely new project, you may want to make use of Benerator's Maven Project Wizard. Otherwise you need to configure the benerator plugin in your project manually. The minimal configuration in Maven's pom.xml would be:

```
<build>

    ...

    <plugins>

        <plugin>
            <groupId>org.databene</groupId>
            <artifactId>maven-benerator-plugin</artifactId>
            <version>0.8.1</version>
        </plugin>

    </plugins>

</build>
```

In order to make use of a plugin, it must be listed as dependency, e.g. dbsanity4ben and mongo4ben:

```
<dependencies>

      <dependency>
            <groupId>org.databene</groupId>
            <artifactId>dbsanity4ben</artifactId>
            <version>0.9.4</version>
      </dependency>

      <dependency>
            <groupId>org.databene</groupId>
            <artifactId>mongo4ben</artifactId>
            <version>0.1</version>
      </dependency>

      ...
```

When using proprietary database drivers (e.g. Oracle), you need to fetch and store them in your Maven repository manually and stated as depenency in the pom.xml.

The default descriptor path points to the file benerator.xml in the project's base directory. So put your benerator.xml into this directory and invoke from the command line:

```
mvn benerator:generate
```

Voilà!

19.3 Common configuration elements

You may configure certain aspects of benerator behavior, e.g. file encoding and scope in a <configuration> element in your pom.xml's plugin configuration, e.g.:

```
<plugin>
    <groupId>org.databene</groupId>
    <artifactId>maven-benerator-plugin</artifactId>
    <version>0.8.1</version>
    <configuration>
        <encoding>iso-8859-1</encoding>
        <scope>test</scope>
    </configuration>
</plugin>
```

These options are applicable for all goals of the Maven Benerator plugin. Their meaning is:

- scope: the scope of the project dependencies to include in the classpath when running benerator. It can be runtime or test. If left out, it defaults to runtime
- encoding: the file encoding to use by default

19.4 Executing descriptor files

The configuration elements available for descriptor file execution are:

- descriptor: the path to the benerator descriptor file
- validate: turn internal (XML an data model) validations off or on
- dbDriver: the JDBC driver class to use, this setting is provided to benerator as variable dbDriver
- dbUrl: the JDBC driver class to use, this setting is provided to benerator as variable dbUrl
- dbSchema: the database schema to use, this setting is provided to benerator as variable dbSchema
- dbUser: the database user name, this setting is provided to benerator as variable dbUser
- dbPassword: the database user's passord, this setting is provided to benerator as variable dbPassword

You can invoke descriptor file execution by calling the **generate** goal from the command line or your IDE:

```
mvn benerator:generate
```

The db* configuration is available to scripts in your descriptor file as well, e.g. <database url=“{dbUrl}“... />

19.5 Creating database snapshots

Use these <configuration> elements in your pom.xml:

- dbDriver: the JDBC driver class to use, this setting is provided to benerator as variable dbDriver
- dbUrl: the JDBC driver class to use, this setting is provided to benerator as variable dbUrl
- dbSchema: the database schema to use, this setting is provided to benerator as variable dbSchema

- dbUser: the database user name, this setting is provided to benerator as variable dbUser
- dbPassword: the database user's passord, this setting is provided to benerator as variable dbPassword
- snapshotFormat: The format in which to create the snapshot. Supported values: dbunit, sql, xls
- snapshotDialect: When createing snapshots in SQL format, this defines the SQL dialect to use. Available values: db2, derby, firebird, hsql, h2, oracle, postgres, sql_server
- snapshotFilename: The file name to use for the snapshot

Start snapshot creation by invoking the dbsnapshot goal:

```
mvn benerator:dbsnapshot
```

19.6 Creating XML files from XML Schema files

Use these <configuration> elements in your pom.xml:

- xmlSchema:
- xmlRoot:
- filenamePattern:
- fileCount:

Then invoke XML file generation using the **createxml** goal:

```
mvn benerator:createxml
```

19.7 Creating a project assembly

For being able to port a Maven project to a location in which no Maven installation is available or possible, you can make the plugin collect all dependencies in one directory and create a classpath file. Call

mvn benerator:assembly

and you will find the project files, all dependent binaries and a classpath.txt file in the directory target/assembly. The classpath.txt helps you to set up a classpath definition for your target execution environment more easily.

19.8 Extending the classpath

If you need to extend the classpath to libraries different to your project dependencies, you can add them as dependencies to your plugin configuration (this requires Maven 2.0.9 or newer):

```
    <plugin>
        <groupId>org.databene</groupId>
        <artifactId>maven-benerator-plugin</artifactId>
        <version>0.8.1</version>
        <configuration>
            ...
        </configuration>
        <dependencies>
            <dependency>
                <groupId>oracle</groupId>
                <artifactId>ojdbc</artifactId>
                <version>1.4</version>
```

```
            </dependency>
        </dependencies>
   </plugin>
```

19.9 Profile-based configuration

In cooperative software development you are supposed to keep your individual configuration private. E.g. you might have individual database configurations on your local development systems. You can then specify them as profile properties in a Maven settings.xml file in your user directory.

```
<profiles>
    <profile>
        <id>development</id>
        <activation>
             <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <database.driver>oracle.jdbc.driver.OracleDriver</database.driver>
            <database.url>jdbc:oracle:thin:@localhost:1521:XE</database.url>
            <database.user>user</database.user>
            <database.pwd>user</database.pwd>
        </properties>
    </profile>
<profiles>
```

You would then refer them in your pom.xml:

```
<plugin>
    <groupId>org.databene</groupId>
    <artifactId>maven-benerator-plugin</artifactId>
    <version>0.8.1</version>
    <configuration>
        <descriptor>src/test/benerator/myproject.ben.xml</descriptor>
        <encoding>ISO-8859-1</encoding>
        <dbDriver>${database.driver}</dbDriver>
        <dbUrl>${database.url}</dbUrl>
        <dbUser>${database.user}</dbUser>
        <dbPassword>${database.pwd}</dbPassword>
        <dbSchema>${database.user}</dbSchema>
    </configuration>
</plugin>
```

19.10 Attaching the Mojo to the Build Lifecycle

You can also configure the benerator plugin to attach specific goals to a particular phase of the build lifecycle. Here is an example:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.databene</groupId>
      <artifactId>maven-benerator-plugin</artifactId>
      <version>0.8.1</version>
      <executions>
        <execution>
          <phase>integration-test</phase>
          <goals>
            <goal>generate</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This causes the benerator goal 'generate' to be executed whenever integration tests are run. For more information on binding a plugin to phases in the lifecycle, please refer to the Build Lifecycle documentation.

For more information, see Maven's "Guide to Configuring Plug-ins": http://maven.apache.org/guides/mini/guide-configuring-plugins.html

20 Extending benerator

benerator can be customized in many ways. It provides Service Provider Interfaces which you can implement in Java for introducing own behavior. These are

- **Generator** generates attribute data or entities with specific characteristics
- **Sequence** lets you define own random or sequence algorithms
- **WeightFunction** allows you to provide a weight function which determines the probability of a certain value.
- **CumulativeDistributionFunction** allows you to provide a weight function which determines the probability of a certain value.
- **Converter** converts data of one type to another and can be used to support custom data classes, data formats or message formats.
- **Validator** validates previously defined data and tells if it is valid. This is useful for low-knowledge data generation where you have e.g. a validation library or system but little knowledge how to construct valid data. In this case you can generate random data and let the validator decide, which to accept.
- **Consumer** receives the generated data and is typically used to store it in a file or system.
- **EntitySource** allows the user to import predefined entity data, e.g. from files with custom data formats.
- **DescriptorProvider** reads metadata from systems or files and provides them as benerator descriptors. You can define an own DescriptorProvider for e.g. importing generation characteristics from annotations in an XMI file.
- **StorageSystem** provides access to a system for storing and querying data. This would be the interface to implement for connecting to your application, e.g. SAP, Siebel or a custom one.
- **Task** is an interface for executing custom Java code.

You can as well plug into the script frameworks that you are binding. So, for FreeMarker, you can implement custom methods in Java and call them from the benerator generation process.

20.1 Custom Generators

20.1.1 Generator Interface

org.databene.benerator.Generator is the basic Generator interface. It has the following methods:

- **Class<E> getGeneratedType()**: Tells the framework of which Java class the generated values are.
- **void init(GeneratorContext context)**: This is called to complete a Generators configuration and initialize it. If the configuration is not alright, the init() method is expected to throw an InvalidGeneratorSetupException. If the method finishes without exception, the generator has to be in state running. The next invocation of generate() is expected to return a valid product.
- **boolean wasInitialized()**: Tells, if the init() method has already been called. Since Benerator massively uses generator chaining and nesting, this is useful to avoid double initialization for each generator instance.
- **ProductWrapper<E> generate(ProductWrapper<E> wrapper)**: Generates an instance of the generic type E and uses the wrapper provided by the caller to return it to the client. If the method is called in an inappropriate state (created or closed), it throws an IllegalGeneratorStateException. If the generator is not available any more, the method returns null.
- **void reset()**: Resets the generator to the initial state. When called, the Generator is expected to act

as if 'restarted'. After invocation the state has to be available.

- **void close()**: Closes the generator. After invocation the state is unavailable.
- **boolean isThreadSafe()**: Tells if the Generator class can safely be called concurrently by multiple threads.
- **boolean isParallelizable()**: Tells if the Generator can be cloned and each instance be executed with a dedicated single thread.

20.1.2 Generator States

Generators have the following life cycle:

- **created**: The generator is under construction. This may take several steps, since generators need to be JavaBeans. When setup is done, a Generator must be initialized by calling its init() method.
- **running**: Generator construction is done and the generator is available. The user may use the Generator calling the generate() method.
- **closed**: The Generator may become unavailable automatically if its value space is depleted or manually when close() has been invoked. The Generator may be reset to the running state again by calling reset() . When being closed , the generator must be in a state in which it can be safely garbage collected.

20.1.3 Helper classes for custom Generators

It is recommendable to make your custom generator extend one of the following classes:

- **AbstractGenerator**: Implements state handling
- **UnsafeGenerator**: Implements state handling and declares to be neither thread-safe nor parallelizable
- **ThreadSafeGenerator**: Implements state handling and declares to be thread-safe and parallelizable

When deriving a custom generator, prefer delegation to inheritance. This simplifies code maintenance and life cycle handling. Abstract Generator implementations which already implement delegate handling are provided by the following classes:

- **GeneratorWrapper**: Wraps another generator of different product type
- **GeneratorProxy**: Wraps another generator of the same product type
- **MultiGeneratorWrapper**: Wraps multiple other generators, e.g. for composing their products, or arbitrarily choosing one of them for data generationGeneratorWrapper for wrapping a single delegate generator of different product type.

20.2 Custom FreeMarker methods

You can define custom functions in Java which will be called by FreeMarker, e.g. a helloWorld function which could be used like this:

```
<attribute name="greeting" script="{helloWorld(Volker)}"/>
```

See the FreeMarker documentation at http://freemarker.sourceforge.net/docs/pgui_datamodel_method.html for some more details. The Java implementation of the class could be something similar to this:

```
public class HelloWorldMethod implements TemplateMethodModel {

      public TemplateModel exec(List args) throws TemplateModelException {
            return new SimpleString("Hello " + args[0]);
      }

}
```

A descriptor file would need to instantiate the class, before it can be called:

```
<bean id="helloWorld" class="HelloWorldMethod" />
```

Unfortunately, anything you provide in the method call will be converted to a List of Strings, so date or number formatting may be necessary on the descriptor side and String parsing on the Java Method side. If the result type of the method is not a SimpleString the same conversions will be done. so you might need to use strange expressions, e.g. for a method that sums up dates and returns a date:

```
<attribute
    name="lastupdatedtime"
    script="${dateSum(deal.created_time?string('yyyy-MM-dd'),
        deallog._delay)?string('yyyy-MM-dd')}"/>
```

20.3 Custom Sequences

Since a Sequence's primary concern is number generation, a Sequence implementor can focus on number generation. By intheriting from the class Sequence, one inherits the data redistribution feature defined in this class and only needs to implement

```
<T extends Number> Generator<T> createGenerator(
      Class<T> numberType, T min, T max, T granularity, boolean unique);
```

The method needs to be implemented in a way that it creates and returns a new Generator component, which generates numbers of the given numberType with a numerical value of at least min and at most max and a granularity, such that each generated value x is x = min + n * granularity, with n being an integral number. If the caller of this method indicated that it requires unique number generation and the Sequence is not capable of generating unique numbers, the method is requires to throw a org.databene.ConfigurationError.

20.4 Custom WeightFunctions

A WeightFunction specifies a probability density for a numeric parameter. If applied to a number generator it will generate numbers with the same probability density. When applied to collections or arrays, the function will evaluate to the probability of an element by its index number.

For defining your own WeightFunction, you just need to implement the WeightFunction interface:

```
package org.databene.model.function;

public interface WeightFunction extends Distribution {
    double value(double param);
}
```

Attention: Take into account, that the parameter may become zero: When using the Function for weighing the entries of an import file or a list, the function will be called with zero-based indexes as argument. So, if you want to use a 10,000-element CSV-file weighted by a custom WeightFunction, it must be able to produce useful values from 0 to 9,999.

20.5 Custom CumulativeDistributionFunctions

TODO (in a later version of the manual)

20.6 Custom Converters

Custom Converters can be used for supporting custom data types or specific data formats. The Converter interface has the following methods:

- **Class<S> getSourceType()**: Returns the Java class whose instances can be converted
- **Class<T> getTargetType()**: Returns the Java class to which objects are converted
- **T convert(S sourceValue)**: Converts a sourceValue of type S to an object of type T
- **boolean isThreadSafe()**: Tells if a converter can be executed with several concurrent threads
- **boolean isParallelizable()**: Tells if a converter can be cloned and each clone can run with a single dedicated thread

These classes are useful parent classes for a custom Converter implementation:

- UnsafeConverter: Declares to be neither thread-safe nor parallelizable
- ThreadSafeConverter: Declares being thread-safe and parallelizable

Beyond this, a custom Converter should

- provide a public default (no-arg) constructor
- exhibit each relevant property by a public set-method

20.7 Custom Validators

A validator must implement at least one of these two interfaces:

- javax.validation.ConstraintValidator (as of JSR 303 - Bean Validation)
- org.databene.commons.Validator

Beyond this, a custom validator should

- provide a public default (no-arg) constructor
- make each relevant property configurable by a set-method

20.7.1 javax.validation.ConstraintValidator

For implementing ConstraintValidators, see the Bean Validation documentation.

20.7.2 org.databene.commons.Validator

For implementing the org.databene.commons.Validator interface, a custom validator must implement the method boolean valid(E object) method returning true for a valid object, false for an invalid one.

It is recommended to inherit a custom Validator from the class org.databene.commons.validator.AbstractValidator. If the Validator interface will change in future versions, the AbstractValidator will try to compensate for implementations of the old interface. Thus, a simple validator implementation which checks that an object is not null would be:

```
public class NotNullValidator<E>
        extends org.databene.commons.validator.AbstractValidator<E> {

    public boolean valid(E object) {
        return (object != null);
    }

}
```

20.7.3 Implementing both interfaces

If you inherit a custom Validator from org.databene.commons.validator.bean.AbstractConstraintValidator, it implements both interfaces.

20.8 Custom Consumers

A Consumer is the final destination that receives the data generated by benerator. It implements the Consumer interface with the following methods:

- **startConsumption(ProductWrapper<E> object)** starts processing of a data object (usually an Entity)
- **finishConsumption(ProductWrapper<E> object)** finishes processing of a data object.
- **flush()** forces the consumer to forward or persist its data.
- **close()** closes the consumer

Consumers must be thread-safe.

Beyond this, it should

- provide a public default (no-arg) constructor
- make each relevant property configurable by a set-method

If your data format and generation supports nesting, the methods startConsuming() and finishConsuming() are called in a hierarchical manner. So, if A contains B, the invocation sequence is:

```
startConsuming(A)
startConsuming(B)
finishConsuming(B)
finishConsuming(A)
```

For non-hierarchical processing you only need to implement the startConsuming() with data processing logic.

20.9 Custom EntitySources

For defining a custom EntitySource you need to implement two methods:

- Class<Entity> getType(): Must return Entity.class
- DataIterator<E> iterator(): Must return a DataIterator which iterates over entities. If the iterator requires resource allocation, it should free its resources on close() invocation.

Beyond this, a custom EntitySource should

- provide a public default (no-arg) constructor
- make each relevant property configurable by a set-method

20.10 Custom Tasks

For executing custom Java code, the most efficient choice is to write a JavaBean class that implements the interface org.databene.task.Task.

The Task interface consists of callback interfaces for execution logic and lifecycle management.

- **String getTaskName()**: returns a task name for tracking execution. On parallel execution, the framework appends a number to identify each instance.

- **TaskResult execute(Context context, ErrorHandler errorHandler)**: implements the core functionality of the task. Tasks may be called several times subsequently and the task uses return valueto indicate whether it is running, has just finished or is unavailable.
- **void pageFinished()**: is called by the framework to inform the task that a 'page' of user-defined size has finished. This can be used, e.g. for grouping several execute() steps to one transaction.
- **void close()**: is called by the framework to make the task close all resources and prepare to be garbage-collected.
- **boolean isParallelizable()**: Tells if the task can be executed in several threads concurrently. This is not used in Benerator.
- **boolean isThreadSafe()**: Tells if it is possible to create clones of the task and execute each clone in a dedicated single thread. This is not used in Benerator.

Benerator provides several implementations of the Task interface which are useful as parent class for custom implementations:

- AbstractTask is a simple abstract implementation that provides useful default implementations for the lifecycle-related methods.
- TaskProxy can be used to wrap another Task object with a custom proxy and add extra functionality.
- RunnableTask can be used to wrap an object that implements the Runnable interface.

A task has access to all Java objects and entities in the context and may arbitrarily add own objects (e.g. for messaging between different tasks). A common usage pattern is to share e.g. a transactional database by the context, have every thread store objects in it and then use the pager to commit the transaction.

21 Using Benerator as Load Generator

Benerator has done the first implementation steps towards being itself used as load generator, not only data generator for preparing load test data.

The basic idea behind the approach is to use a <generate> element for composing invocation parameters, and use special 'Consumer' or 'Task' implementations for doing the actual invocation of the system under test.

Performance of task execution can be tracked, setting a stats attribute to true:

```
<run-task class="com.my.TaskImpl"
    invocations="10000" threads="100"
    stats="true" />
```

The performance of consumer invocations can be tracked, using the PerfTrackingConsumer class:

```
<import platforms="contiperf" />

<bean id="myWS" spec="new MyWebServiceConsumer(...)"/>

<generate type="params" count="10000">

    <value type="int" min="100" max="10000" distribution="random"/>

    <consumer spec='new PerfTrackingConsumer(myWS)'/>

</generate>
```

21.1 JavaInvoker

To aid you calling Java code from Benerator, there is a Helper class, JavaInvoker, which implements the Consumer interface and calls a Java method, which is declared by its name, on a Java object. The data of the generated entity is automatically:

```
<bean id="service" spec="..." />

<bean id="invoker" spec="new JavaInvoker(ejb, 'enrolCustomer')" />
```

For tracking invocation performance, you need to add the PerfTrackingConsumer:

```
<bean id="enrolCustomer" class="PerfTrackingConsumer" >

    <property name="target" value="{invoker}" />

</bean>
```

21.2 Checking performance requirements

You can as well define performance requirements using properties of the class PerfTrackingConsumer:

Property	Description
max	the maximum number of milliseconds an invocation may take
percentiles	a comma-separated list of percentile requirements in the format used in ContiPerf

By default, execution times are printed to the console. You can as well plug in custom ExecutionLoggers for saving log data to file or feed it to other applications. This is done using the PerfTrackingConsumer's executionLogger property.

An example for the properties described above:

```
<import class="org.databene.contiperf.log.ConsoleExecutionLogger" />
```

```
<bean id="enrolCustomer" class="PerfTrackingConsumer" >
    <property name="target" value="{invoker}" />
    <property name="executionLogger" value="{new ConsoleExecutionLogger()}" />
    <property name="max" value="5000" />
    <property name="percentiles" value="90:5000, 95:7000"/>
</bean>
```

22 Troubleshooting

22.1 (Out of) Memory

If you get an OutOfMemoryError, first increase the Java heap size by an -Xmx environment setting (e.g. by adding -Xmx1024m to the BENERATOR_OPTS in the script files).

Another potential cause for OutOfMemoryErrors is application of distributions to very large data sets. Most sequences and all other types of distribution require the source data to fit into RAM. So either use an 'unlimited' sequence like 'expand', 'repeat' or 'randomWalk' or simply repeat data set iteration by adding cyclic="true" to the configuration.

22.2 temp Directory

On some environments, the temp directory has a very restrictive disk quota. If you need more space for data generation, you can specify another directory by the -Djava.io.tmpdir environment setting (e.g. by adding -Djava.io.tmpdir=/User/me/mytemp to the BENERATOR_OPTS in the script files)

22.3 File Encoding

If no file encoding was specified, benerator uses the default file encoding of the system it runs on - except if the file itself contains encoding info (like XML).

If all used files have the same encoding and it is different to your system's encoding, you can change set benerator's default encoding by the -Dfile.encoding environment setting (e.g. by adding -Dfile.encoding=iso-8859-1 to the BENERATOR_OPTS in the script files)

When generating data in heterogeneous environments, it is good practice to set the defaultEncoding property of the benerator descriptor file's root element. If only single files have a different encoding, you can specify an encoding properts for all built-in file importers and file-based consumers.

A typical error that may arise from wrong file encoding configuration is that file import (e.g. for a CSV file) stops before the end of file is reached.

22.4 Logging

benerator logs its event using apache commons-logging. That service forwards output to Apache log4j or to the native JDK 1.4 logging. For avoiding version conflicts with your environment, benerator uses JDK 1.4 logging by default, but for troubleshooting it is useful to switch to Log4j as the underlying logging implementation and fine-tune log messages for tracking down your problem. In order to use log4j, download the binary of a new version (e.g. log4j 1.2.15) from the Apache log4j 1.2 website, uncompress it and put the jar file log4j-1.2.15.jar into benerator's lib directory. Edit the log4j.xml file in your BENERATOR_HOME/bin directory to adapt the log levels for interesting categories:

Set a category to debug for getting detailed information about its execution. The most important log categories are:

name	description
org.databene.benerator.main	Events of benerator's main classes, e.g. detailed information about which entities are currently generated
org.databene.benerator.STATE	generator state handling for information which component generator caused termination of the composite generator
org.databene.benerator.factory	Creating generators from descriptor information
org.databene.benerator	Top-level directory for all generators and main classes

org.databene.SQL	SQL commands, e.g. DDL, queries, inserts, updates
org.databene.JDBC	JDBC operations, e.g. connection / transaction handling
org.databene.jdbacl.model.jdbc	Database meta data import
org.databene.platform.db	All database related information that does not fit into the SQL or JDBC category
org.databene.platform.xml	XML-related activities
org.databene.domain	benerator domain packages
org.databene.model descriptor	related information
org.databene.commons	low-level operations like data conversion

22.5 Locating Errors

When configuring data generation you are likely to encounter error messages.

Depending on the settings it may be difficult to find out what caused the problem. For tracking database-related errors, set batch="false" in your <database> setup and use pagesize="1" in the <generate>. These are default settings, so you do not need to specify them explicitly if you did not change the default.

If that alone does not help, set the log category org.databene.benerator.main to debug level to find out which element caused the error. If there is a stack trace, check it to get a hint which part of the element's generation went wrong. If that does not help, remove one attribute/reference/id after the other for finding the actual troublemaker. If you still cannot solve the problem, post a message in the benerator forum. You can check out the benerator sources from the SVN source repository, open it in Eclipse and debug through the code.

22.6 Database Privilege Problems

When importing database metadata, you might encounter exceptions when Benerator tries to get metadata of catalogs or schemas it has no access privileges to.

Usually can fix this by choosing the right schema for your database, e.g.

```
<database id="db" ... schema="PUBLIC" />
```

If you are not sure which schema is applicable in your case, edit the logging configuration in log4j.xml (as described above) and set the category **org.databene.platform.db** to **debug**.

You will then get a list of schemas as Benerator scans the database metadata, e.g. for an Oracle system:

```
06:03:45,203 DEBUG [DBSystem] parsing metadata...

06:03:45,203 DEBUG [JDBC] opening connection to
jdbc:oracle:thin:@10.37.129.3:1521:XE

06:03:45,226 DEBUG [JDBC] Created connection #4:
oracle.jdbc.driver.T4CConnection@741827d1

06:03:45,385 DEBUG [JDBC] opening connection to
jdbc:oracle:thin:@10.37.129.3:1521:XE

06:03:45,417 INFO  [JDBCDBImporter] Importing database metadata. Be patient,
this may take some time...

06:03:45,417 DEBUG [JDBCDBImporter] Product name: Oracle

06:03:45,419 INFO  [JDBCDBImporter] Importing catalogs

06:03:45,430 INFO  [JDBCDBImporter] Importing schemas

06:03:45,438 DEBUG [JDBCDBImporter] found schema ANONYMOUS
```

```
06:03:45,438 DEBUG [JDBCDBImporter] found schema DBSNMP
06:03:45,438 DEBUG [JDBCDBImporter] found schema DIP
06:03:45,438 DEBUG [JDBCDBImporter] found schema FLOWS_FILES
06:03:45,439 DEBUG [JDBCDBImporter] found schema FLOWS_020100
06:03:45,439 DEBUG [JDBCDBImporter] found schema HR
06:03:45,439 DEBUG [JDBCDBImporter] found schema MDSYS
06:03:45,440 DEBUG [JDBCDBImporter] found schema OUTLN
06:03:45,440 DEBUG [JDBCDBImporter] found schema SHOP
06:03:45,440 DEBUG [JDBCDBImporter] importing schema SHOP
06:03:45,441 DEBUG [JDBCDBImporter] found schema SYS
06:03:45,441 DEBUG [JDBCDBImporter] found schema SYSTEM
06:03:45,441 DEBUG [JDBCDBImporter] found schema TSMSYS
06:03:45,441 DEBUG [JDBCDBImporter] found schema XDB
```

Cross checking this with your access information should make it easy to figure out which one is appropriate in your case.

See the Usual Database Settings.

22.7 Constraint Violations

Some constraint violations may arise when using database batch with nested create-entities. Switch batch off. If the problem does not occur any more, stick with non-batch generation. Otherwise you need further investigation. When using Oracle, a constraint violation typically looks like this:

```
java.sql.SQLException: ORA-00001: Unique Constraint  (MYSCHEMA.SYS_C0011664) violated
```

It contains a constraint name you can look up on the database like this:

```
select * from user_constraints where constraint_name like  '%SYS_C0011541%'
```

The query result will tell you the table name and the constraint type. The constraint types are encoded as follows:

P: Primary key constraint

U: Unique constraint

R: Foreign key constraint

22.8 'value too large for column' in Oracle

Depending on the character set, oracle may report a multiple of the real column with, e.g. 80 instead of 20. So, automatic generation of varchar2 columns may fail. This typically results in Exceptions like this:

```
java.sql.SQLException: ORA-12899: value too large for column "SCHEM"."TBL"."COL" (actual: 40, maximum: 10)
```

This is Oracle bug #4485954, see http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/readme_jdbc_10204.html and

http://kr.forums.oracle.com/forums/thread.jspa?threadID=554236.

The solution is using the newest JDBC driver, at least 10.2.0.4 or 11.0. BTW: It is backwards compatible with the Oracle 9 databases.

22.9 Time is cut off in Oracle dates

Oracle changed back and forth the mapping of internal types to JDBC times in Oracle 8 and 11, the mapping in Oracle 9 and 10 is wrong, see http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-faq-090281.html#08_00. In order to fix the behaviour, use the newest available Oracle 11 JDBC driver you can get, it is backwards compatible down to Oracle 9 and provides a proper mapping of the date type for all Oracle database versions.

22.10 Composite Keys

Benerator expects single-valued ids. It does not automatically support composite keys and composie references

Since composite keys typically have a business meaning, most composite keys cannot be automatically generated. So there is no need to support this.

If you encounter a composite key, manually configure how to create each key component.

22.11 Importing Excel Sheets

For a beginner it is sometimes confusing, how Benerator handles imported Excel sheets. For this task iit completely relies on the cell type configured in the original sheet. So if you have a Date cell in the Excel sheet and format it as number or text, benerator will interpret it as double or string.

Another popular error comes from columns that contain long code numbers and have the the default format: They are imported as numbers and e.g. leading zeros are lost. In such case explicitly format the column as text in Excel.

Apache POI represents all numbers as variablesof type 'double'. So there are numbes which are simple in decimal format but not so in binary: when importing the number 1.95 from an Excel sheet, the user gets a value of 1.95000000002. For now you need to round the values yourself, e.g. by a converter.

22.12 Number Generation on Oracle

When generating very large decimal values (>> 1.000.000.000) in an Oracle database you may observe that smaller numbers are written to the database, losing some trailing zeros or even cutting the whole number to a decimal with one prefix digit and several fractional digits. This results from a bug in Oracles older JDBC drivers and can be fixed by using the newest driver version (note that you can even use 11.x JDBC drivers for 10.x databases).

22.13 Unknown Column Type

If your application makes use of a database's proprietary column types, you may run into an exception when Benerator encounters it. If you know how to create and handle data for this column type, you can do so by configuring the database to accept unknown column types :

```
<database ... acceptUnknownColumnTypes="true">
```

22.14 Table 'X' not found in the expected catalog 'Y' and schema 'Z'

This message tells you, that you database configuration is wrong. Check and fix the 'schema' and 'catalog' settings in your database configuration, e.g.

```
<database … catalog="main" schema="Z" />
```

or your environment configuration, e.g. xyz.env.properties:

```
db_catalog=main
db_schema=Z
```

Note: On most systems (e.g. Oracle, HSQL) no catalog needs to be specified.

23 Monitoring Benerator

You can monitor Benerator using a JMX client, for example JConsole.

The following properties can be monitored:

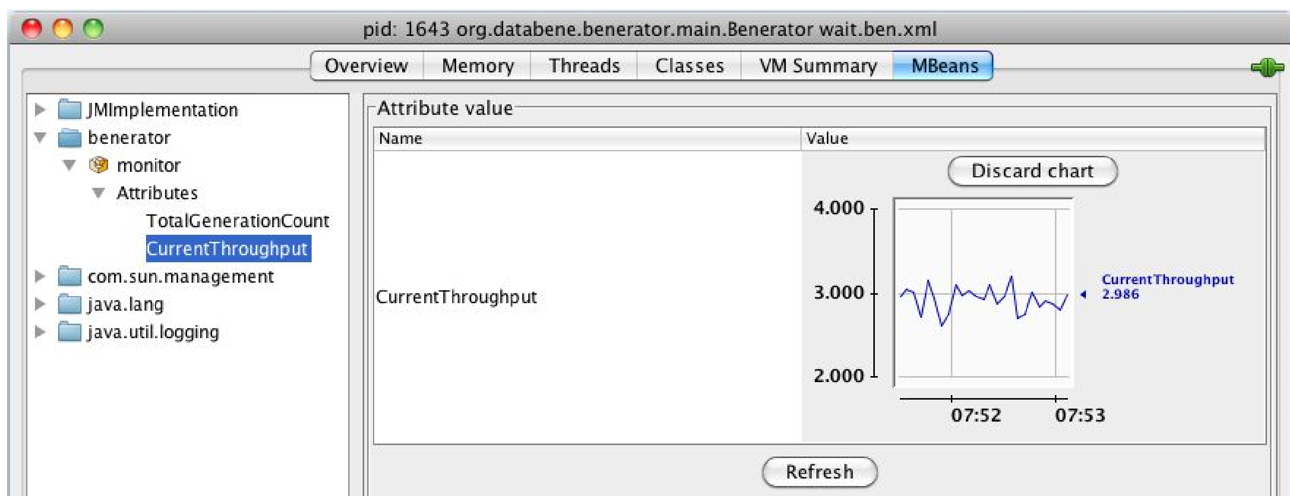
Property	Description
TotalGenerationCount	The total number of generated data sets
CurrentThroughput	The number of data sets generated per second
OpenConnectionCount	The number of currently open database connections
OpenResultSetCount	The number of currently open database query result sets
OpenStatementCount	The number of currently open database statements
OpenPreparedStatementCount	The number of currently open prepared database statements

The first two properties, **TotalGenerationCount** and **CurrentThroughput**, are used for Benerator performance monitoring and optimization. If you suspect Benerator to be 'hanging', first check its **CurrentThroughput**.

The last four properties (**Open...**) for database resource monitoring and database resource leak detection.

23.1 Monitoring with JConsole

1. Start JConsole on the command line
2. Select a process
3. Choose the MBeans tab
4. In the left tree view, select
5. benerator – monitor – Attributes
6. Select the attribute TotalGenerationCount or CurrentTroughput and the value is displayed on the right
7. Double clicking the number opens a chart that displays the value's evolution over time



23.2 Remote monitoring

For monitoring Benerator execution from a remote machine, you need to set some BENERATOR_OPTS. Here are only the simplest and basic settings.

Warning: The settings described here do not provide any security and thus are recommended only for evaluation! If you do want to monitor a sensitive system in a remote manner, you need to apply security settings as described in http://download.oracle.com/javase/1.5.0/docs/guide/management/agent.html!

These are server-side settings and are independent of the client you are using:

Option	Description
-Dcom.sun.management.jmxremote	Enable remote access
-Dcom.sun.management.jmxremote.port=9003	Configures the port for remote access
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false	Enables anonymous and unsecure access (not recommended)

24 Benerator Performance Tuning

24.1 Performance

benerator is regularly optimized for performance, so single threaded data generation is efficient enough in the most cases. As a result, multithreaded generation is tested much less intensively than singlethreaded operation. The following recommendations are ordered in 'bang-for-the-bucks' order: The first tips have a better ratio of effort to gain than the later ones.

24.2 pageSize

By default benerator stores each entity in an own transaction, because this simplifies error tracking when creating a descriptor file. But this has a tremendous impact on performance. When configuration is complete, and you need performance, set the pagesize attribute of critical <generate> elements, e.g. <generate type="db_user" count="1000000" consumer="db" pagesize="1000">

24.3 JDBC batch

JDBC batches provide for significantly better database insertion performance than standard operation. In benerator this is turned of by default, since error messages that arise from bad generation setup are much harder to analyze in batch mode. When you are finished with defining data generation and need performance for mass data, you can activate batch operation by the batch attribute of the database element:

```
<database ... batch="true" />
```

benerator is optimized for performance. Thus you may get problems when combining nested <generate> elements with batching. It typically results in exceptions that indicate violation of a foreign-key constraint.

24.4 Query fetch size

If you are querying data with large result sets from your database, e.g. when anonymizing production data, you should tune the database's fetch size. It determines how many rows are transmitted from the database to benerator when accessing the first query result. So it reduces the number of network roundtrips. benerator uses a fetch size of 100 by default which should be useful in most cases.

You can experiment with higher values, e.g. 1000 by configuring the batch attribute of the database element: <database ... batch="true" />. This is mainly useful if the database is accessed over a slow network connection and query result sets are at least as large as the fetch size and are iterated to a relevant extent. When setting the fetch size to value that is too high, performance may actually decrease.

24.5 Id Generation

The most efficient id generation strategy is increment since it works without connecting the database. It works fine for multithreaded generation, too. But for concurrent execution of multiple benerator processes or continuation of a cancelled generation process you need an id generation that is unique among several runs. The most efficient id strategies with such behavior are seqhilo (database-based) and uuid (universally unique string id).

Relational Data Generation

Nesting <create-entities> elements greatly improves maintanability of data generation but can decrease generation speed. At critical generation steps you might want to sacrifice maintainability for speed by replacing nested structures with 'relational' structures, e.g. replacing this code:

```
<generate name="db_customer" count="1000000" consumer="db">
    <generate name="db_order" minCount="0" maxCount="20 " consumer="db">
        <attribute name="customer_fk" script="{db_customer.id}"/>
        ...
```

```
    </generate>
</generate>
```

with something like this:

```
<generate name="db_customer" count="1000000" consumer="db" />
<generate name="db_order" count="10000000" consumer="db">
   <reference name="customer_fk" source="db" targetType="db_customer" distribution="random" />
    ...
</generate>
```

24.6 Scripts

When using other script languages that DatabeneScript, the scripts expressions are parsed at runtime, thus are significantly slower than compiled code. For performing CPU-intensive operations or excessive looping, use DatabeneScript or program a Java task (See Manual Section 10, “Custom Tasks”).

24.7 Parsing (Oracle) metadata

On databases with many tables, scanning metadata can take several minutes. One source of superfluous tables is the Oracle recyclebin. You can speed up parsing by including a 'purge recyclebin' in your SQL scripts.

Using regular expressions in the <database>'s excludeTables and includeTables settings, you can restrict the amount of metadata to be parsed.

If metadata retrieval still takes too long, you can use <database... metaCache="true" /> for storing metadata in a cache file on your local hard drive.

When Benerator executes SQL with an <execute> statement, it analyzes if the database structure is modified. In that case, the cached database meta data is invalidated and reparsed when they are needed the next time. If you are certain that the change is irrelevant to subsequent generations steps, you can suppress the invalidation by

```
<execute invalidate="false">ALTER TABLE...</execute>.
```

24.8 Distributed Execution

You can distribute generation over several machines. In oder to do so you will need two types of descriptor files: First an initialization descriptor that initializes the systems and generates the deterministic core data, Second a mass data description file. The initialization file will need to run first on a single machine, then the mass data file can be executed on multiple processes or machines concurrently. For mass data generation you will need to take special care: Choose an id generation strategy that is able to create unique ids under these circumstances (see Section 24, “Generating IDs”).