

2003/3 XCGF

Holger Zahnleiter
holger@zahnleiter.org

Date: 13.05.03

Abstract

The Extensible Code Generator Framework (XCGF) is a Java class library providing classes, which ease the creation of code generators and which are essential for code generator development. This framework is primarily intended for generating program code on IT projects, e.g. frame classes containing properties or data access classes. It was not intended to dynamically generate HTML or XML documents on a web server. It was not tested for such load situations.

1 Project characteristics

Project code	2003/3
Project name	XCGF
Started	01.04.03
Ended	not yet
Version/release	1/1
Used tools and libraries	JDK 1.4.1 (sould also work with older versions of JDK); JUtil (my Java library, available on my homepage); JKVPF (available on my homepage); Xerces 2.0.1;
Runs on platform	Any system with a JVM

2 Release notes

2.1 Version 1, release 1

- Input providers for XML text files and JDBC datasources.
- Building blocks for text output.
- Output facility for text files (no indentation, yet).
- Generator frames for SAX and DOM input. Generator frame, which walks the DOM tree in pre-order and calls user-defined methods for each node.
- Example generator.

3 Introduction

Why is it called extensible and what is special about this framework?

What is special about this framework is, that it creates an abstract and general model of code generators and introduces several abstractions on generator **input**, generator **output** etc. As a direct conclusion it features the concept of the **providers**. Providers can access data sources of

different type, such as text files, databases or binary files and transform them to XML. Doing so creates an uniform view on the input data – XML. In general, code generators are depending on the input provided to them. Their output differs for different input.

The framework is called extensible because of two characteristics. First, extensibility partially comes from being "just" a class library, you can create whatever code generator you want. Second, the framework introduces abstractions (and implementations) for typical code generator components, especially the providers. You can extend the framework by building your own problem specific providers and implementations of other abstract code generator components.

4 Code generator model

The picture below is the visualisation of a model for an abstract code generator, which stands behind the XCGF. In this paper, a code generator is seen as a program, which takes three different kinds of input to produces its output.

The yellow boxes are representing abstract components, provided by the framework. The blue boxes are implementations of that abstract components, also included with the framework. Anything else has to be implemented for a specific problem by deriving new classes from abstract components or their implementations. The code generator itself is half white and half yellow. This is, because there is a abstract generator, already providing some basic functionality. But in general, you have to derive and write your own generator appropriate to your specific problem.

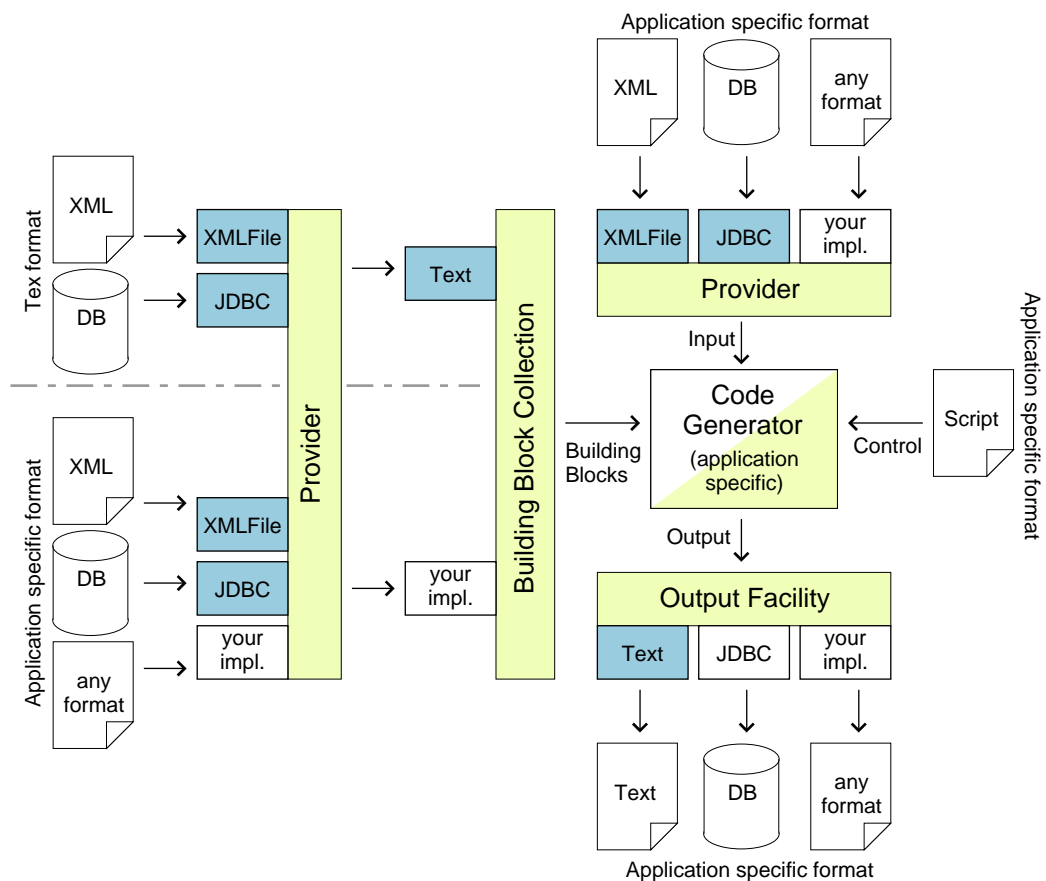


Figure 1: Abstract model of a code generator

- **Providers:** Let's start the explanation of the above figure with the most essential concept of the frame work: the providers. Providers can be seen as general datasources, which present

all kinds of data to the other framework components in an unified way: XML. What ever data is required by the framework components, they can retrieve it by using a provider and access the data with DOM or XPath navigation.

Why was XML chosen as a representation for all kinds of input data? One could have chosen a proprietary tree implementation. This would also allow DOM- or XPath-like addressing of input data elements. The decision is manifold. It's not only the tree-like structure. It's also the possibility to transform atomic pieces of input information to a standardised and language independent way by applying XML schema types such as date, integer, string etc. Also, one can use his XML parser of choice for processing input data.

- **Building blocks:** Everytime the code generator produces (a piece of) output, it will apply one of the given building blocks. The name "building block" originates from the original idea of generating program code, which is a text file most likely. Therefore, "template" would have also been an appropriate term. Indeed, there is a concrete implementation of that abstraction shipped with the framework, representing text snippets, from which the generator can create source code from. What is common to all kinds of building blocks is, that each block within a set of building blocks has an unique name and might have a set of **variables**. The variables receive their value from the input the moment a block gets applied.

In the picture, you see the building block area (left side) divided into two sections. The upper one is representing the predefined text type building blocks. Below you see the kind of building blocks which might be implemented by you. Please note, that for the text building block there is a predefined resource format. This format allows to define text snippets, give them an unique name and define a set of variables.

Let's loose a few more words about the building blocks. Why is there a layer of abstraction, why not just providing text type blocks? Being abstract allows not only to generate text files. As you see, the blocks are getting passed to the output layer, which is responsible for generating the output. A text output component will generate sourcecode files. But there might be other output components, which produce output, which is not a text file. In that case blocks which are not text might be more appropriate. The output component might directly connect to a database system and create tables, indexes etc. from the information provided by the building block objects. In this scenario, building blocks are commands and the output component is a command processor¹.

- **Input:** The input is the kind of data, which makes the difference between the different instance documents (output) generated by the generator. As mentioned already, the blocks passed to the output might have variable parts, called variables. When the output gets constructed by combining building blocks in a particular order, the variable parts need to be set with values. These values are taken from the input. Atomic input elements are transferred to variables of the building block which has to be applied next.

An example might clarify this some more. Assume the input contains class names. Assume furthermore, that the generator generates Java classes. In that case building blocks of type text would fit the best. There would be a block called "beginClass". This block would have a variable called "className": `public class <className> {`. Now the variable gets set with a value from the input: "Customer". The resulting text snippet to go to the output would then be: `public class Customer {`.

- **Control:** The control script is used by the generator to control its workflow – it controls the process of code generation. It could be something like a XSLT script. It is highly application specific. The generator framework does not make any assumptions about the control component (yet).
- **Output:** The output is the result, generated by the code generator. Usually it's text, which gets stored as a file or in a database maybe. But as we have seen already there is an abstraction layer for output components. Basically, an application specific output component takes application specific building block objects and generates application specific output from that.

¹The abstract output layer allows creation of a complete code generator for enterprise application development. It can not only create classes and value objects, it can also create database tables and populate them, if necessary.

5 Providers

Providers are used by both, building block collections and input² to load a resource file. But also the control script might be loaded using a provider. Since all components seem to be based on providers, we will start our explanations with them.

5.1 Class ProviderFactory

You can't directly create a provider. You need to ask the provider factory to do that for you. By doing so, you pass the provider's **registry alias** and **resource descriptor**³ to the factory. When the factory creates the provider it passes the descriptor on to the newly created provider, so that it can access the resource in question.

The registry alias is an unique and symbolic name for the provider's "real" name — the fully qualified class name. With XCGF's **configuration** component you can register custom tailored providers on your own. XCGF has a mapping table (registry), which maps aliases to class names before loading the provider. Using an alias frees you from using long class names. We will hear about the configuration component later in this paper.

The resource descriptor is a list of key/value pairs, separated by semicolons. In case of a provider for XML text files it might look as simple like this: "file=c:/test/input.xml". In case of a database provider, things might become more complex. You might need a datasource name, user name, password etc.

Use the following methods of the `ProviderFactory` class to get yourself a provider factory and to ask that factory for a provider of your choice, respectively.

```
public static ProviderFactory getFactory( Configuration cfg )
throws XCGFProviderException;
public Provider createInstance( String registryAlias, String resourceDescriptor )
throws XCGFProviderException;
```

5.2 Class Provider

The interface of a provider is a quite simple one. Both methods return the content from a data-source. The first as an `InputStream` for a SAX parser, the latter as a DOM tree for a DOM parser.

```
public abstract InputStream getXMLForSAX() throws XCGFProviderException;
public abstract Document getXMLForDOM() throws XCGFProviderException;
```

But how does the provider know, what data to return? Remember the resource descriptor passed to it when constructed by the factory! When implementing your own provider you have to implement the following method. This sets up the parsing framework so that a resource descriptor can be parsed and understood. The values from the descriptor are available by calling `protected Object getConnectInfo(String key)` and applying the parameter name as key.

```
protected abstract KeyValueTracker setUpParsingFrameworkSpecific()
throws Exception;
```

Another method you have to implement is this one. It's purpose is to do a cleanup and free system resources after a provider did it's job and isn't needed anymore. The framework also includes a generator base class you might extend, it already calls `close()`.

```
protected abstract void close() throws XCGFProviderException;
```

²Basically true: Input = Provider.

³All resource descriptors are lists of key/value pairs, separated by semicolon. I use my Key/Value Parsing Framework to process these expressions. See the JKVPF documentation available on my homepage for more details.

5.3 Class Provider4XMLFile

This is one of XCGF's default providers. Task of that provider is to access a XML text file, stored in your local file system (no http or ftp). It is registered under `XMLFile`. So any time you need a provider of that type, use `XMLFile` as the registry alias when asking the factory to create a provider.

Registry alias:

`XMLFile`

Resource descriptor:

```
<connect info> ::= <file>
<file> ::= 'file' '=' any string
```

5.4 Class Provider4JDBC

Another default provider of XCGF is JDBC. It's purpose is to provide content from databases to other XCGF components.

Registry Alias:

`JDBC`

Resource Descriptor:

```
<connect info> ::= <driver><url> ( (<view><mode>[<documentAlias>][<recordAlias>]) |
[<select>] )
<driver> ::= 'driver' '=' any string
<url> ::= 'url' '=' any string
<view> ::= 'view' '=' any string
<mode> ::= 'mode' '=' ( condensed | flat )
<documentAlias> ::= 'documentAlias' '=' any string
<recordAlias> ::= 'recordAlias' '=' any string
<select> ::= 'select' '=' <select expression>
<select expression> ::= <parent table> '[' <join expression> ']'0..*
<join expression> ::= <parent key> '->' <child key> ':' <select expression>
```

The resource descriptor starts with the JDBC driver. This might be for example the JDBC to ODBC bridge `sun.jdbc.odbc.JdbcOdbcDriver`. This driver class gets loaded and therefore needs to be in the classpath.

The second parameter is the JDBC URL, this is a qualifier, which tells the provider which driver (from all loaded ones) to use and which datasource to connect. An example is `jdbc:odbc:JDBCTest`. This will access the ODBC datasource `JDBCTest` via the JDBC to ODBC bridge. Before that, you must have set up the datasource with the JDBC admin tool.

Now, you can choose between two modes. If you give the parameter `view`, then the provider assumes, that you access a single table or view. Using the parameter `select` instead offers you the possibility to enter a select statement and access multiple tables/views.

View mode: In the view mode the only table/view accessed is the one given by the `view` parameter. The provider accesses all records and columns and creates a XML document from it. With the `mode` parameter you can choose whether data gets put to the XML document as read from the database (`flat`) or that columns belonging to the same table are getting concentrated under the same parent XML node record-wise. The table below represents two records returned from a view, where the column names are read like this: `<table name>.<column name>`.

Field.Name	Type.Name	Type.Default
FirstName	String	Darth
LastName	String	Vader

Table 2: View mode example flat vs. condensed

With `mode=flat` the XML document extract produced by this provider for the first record would look somehow like this:

```
<record>
  <Field>
    <Name>FirstName</Name>
  </Field>
  <Type>
    <Name>String</Name>
  </Type>
  <Type>
    <Default>Darth</Default>
  </Type>
</record>
```

With `mode=condensed` the XML document extract produced by this provider for the second record would look somehow like this:

```
<record>
  <Field>
    <Name>LastName</Name>
  </Field>
  <Type>
    <Name>String</Name>
    <Default>Vader</Default>
  </Type>
</record>
```

By default, documents produced by this provider are surrounded by `<document>` tags. Records are surrounded by `<record>` tags. You can change these names by applying the parameters `documentAlias` and `recordAlias`.

Select mode: The structure of the `select` parameter is a bit more sophisticated than a simple name as seen before. It's not really a select expression as you are used to when using databases. It's a simplified select, which only offers to choose a table and other related tables. Lets stick with the above example. We have the `Field` and the `Type` tables. Let's assume each table has a column for an unique key, called ID. The relation between `Field` and `Type` is implemented by a `TypeID` column in `Field` holding foreign keys "pointing" to entries from the `Type` table. A "select" expression would then look like this: `select=Field[TypeID->ID:Type]`. You can have more tables joining `Field`: `select=Field[] [] [] ...` and each joined table can have it's own table(s) joined: `select=Field[TypeID->ID:Type[XSDTypeID->ID:XSDType]] [] [] ...`

Taking the above example table, the following XML document would be created by this provider:

```
<Field-list>
  <Field>
    <Name>FirstName</Name>
    <Type-list>
      <Type>
        <Name>String</Name>
        <Default>Darth</Name>
      </Type>
    </Type-list>
  </Field>
  <Field>
    <Name>LastName</Name>
    <Type-list>
      <Type>
```

```

        <Name>String</Name>
        <Default>Vader</Name>
    </Type>
</Type-list>
</Field>
</Field-list>

```

As you see, the content of each joined table is introduced by surrounding `<table name"-list">` tags, eventhough there might only be one joined element. This is a direct result of the simple straight forward approach.

A custom value mapper `SelectValueMapper` was created to parse the `select` parameter's value. It parses and converts such select expression by constructing an instance of a inner class called `SelectExpression`. This class represents select expressions. Essentially `SelectValueMapper` leaves everything up to `SelectExpression`. `SelectValueMapper`'s only method `map()`—looks like this:

```

public Object map( String expression )
throws Exception
{
    return new SelectExpressionNode( expression );
}

```

An inner class wasn't necessary. The functionality could have also been placed within `SelectValueMapper` itself. `map()` would have looked like this then:

```

public Object map(String expression)
throws Exception
{
    return new SelectValueMapper( expression );
}

```

It was a philosophical decision to implement it the way it is. The mapper object is meant to convert an expression of any kind (string) into an object, which can be seen as the in-memory representation or object model of that expression. I would consider returning a mapper object impure. Never mind.

What is more important is, that the result returned by `map()` is an object graph, which equals the select expression in structure. This object graph is also capable of retrieving the information from the database and returning it as a DOM tree, which reflects the data structure as defined by the select expression. So, for the select mode, the only thing the JDBC provider needs to do, is to use the mapper to create the object tree from the `select` parameter's value and ask that tree for the XML document — done!

6 Building Blocks

Similar like with providers, you don't create building blocks directly. Each block belongs to a collection of building blocks. You first ask the building block collection factory to create such a collection for you, then you can ask that collection for particular blocks.

6.1 Class BuildingBlockCollectionFactory

```

public static BuildingBlockCollectionFactory getFactory( Configuration cfg )
throws XCGFBuildingBlockException;

```

is used to access the collection factory. Again, a configuration object is required. Similar like providers, also different building block types are registered and can be accessed via a registry alias. You can also register custom block types. This registry is part of the configuration. We will discuss that later. Once you have acquired the factory you can use it to query building block collections.

```

public BuildingBlockCollection createInstance( String registryAlias,

```

```
String resourceRegistryAlias, String resourceDescriptor )
throws XCGFBuildingBlockException, XCGFProviderException;
```

`registryAlias` determines which building block type (class) you want to use. The second parameter, `resourceRegistryAlias`, tells, which provider type shall be used to load the collection. Basically collection resources can be seen as serialised collections. You can store them as files in different formats or in a database etc. `resourceDescriptor` is used by the provider to locate and access the resource.

To create your own pair of building block and building block container, you need to implement the following method. This loads (de-serialises) the collection by loading single blocks from the datasource. Which datasource type (provider) to access is told by the `resourceRegistryAlias`, the datasource itself is described by the `resourceDescriptor`.

```
textttpublic abstract BuildingBlockCollection readFromSource()
throws XCGFBuildingBlockException, XCGFProviderException;
```

6.2 Class BuildingBlockCollection

A building block collection is a number of building blocks. Imagine you want to create Java classes. In that case it would make sense to have a collection consisting of blocks for the beginning and the ending of classes and methods. There would be blocks for defining properties and access methods etc.

Additionally there are different types of building block collections derived from the base class `BuildingBlockCollection`. There are collections which hold blocks for the creation of text files. Other collections might hold blocks for creation and population of database tables.

After a collection of a certain type has been created by the factory, it will be initialised⁴ by calling `init()` and applying the parameters passed to the factory's method `createInstance()`.

```
protected void init( Configuration cfg, String registryAlias,
String resourceDescriptor ) throws XCGFBuildingBlockException,
XCGFProviderException;
```

To use a building block you only need to have this simple interface in mind. The method will return the building block, which matches the given name. This method will only succeed in case the resource file contained a building block with the given name.

```
public BuildingBlock getBuildingBlock( String name )
throws XCGFBuildingBlockException;
```

6.3 Class BuildingBlock

Like most other classes described so far `BuildingBlock` is an abstract base class. The idea behind that class is, that a building block contributes to the output produced by the generator. This contribution is made up of invariant parts as well as variable parts, where the invariant parts are inherent to the block and the variable parts might be set from the input. Additionally, a block needs to have a unique name to be accessible⁵.

⁴All objects created by factories are created by dynamically loading the class. Therefore, the only way to create an instance is by applying the default constructor, which does not take parameters. This, right after creating an object, the factory will call the `init()` method to pass on further information, i.e. configuration, resource descriptor etc.

⁵Objects can also be referenced by pointers/references. They might also be stored in a collection and therefore referenced by ordinal or hash key. But don't forget, that intention of this framework is to create code generators. These programs in many cases are controlled by script. So, access by name to providers, building blocks and other components is required.

The variable elements of a building block are called variables. There are two ways to set their values. First is to set the value of a single variable denoted by variable name. The second is to set all values at once.

```
public void setVariable( String name, Object value )
throws XCGFBuildingBlockException;

public void setVariables( Vector values ) throws XCGFBuildingBlockException;
```

As you have noticed, variables can represent values of any type. Building blocks don't necessarily have to be text snippets.

Another important method of `BuildingBlock`'s interface is `getPreliminaryOutput()`. What does that mean? The output is created by the output facility. So, this method might not return a direct contribution to the output. It can return an object of any type. The only thing you need to have in mind is, that the output facility in charge needs to be able to handle that preliminary output object in order to be able to create real output from it.

```
public abstract Object getPreliminaryOutput()
throws XCGFBuildingBlockException;
```

In case of text blocks, this method might indeed return a string, which directly goes to the output. In other cases it might return a command object, which gets interpreted and executed by the output facility in order to produce output.

6.4 Class `BuildingBlock4Text`

Currently there is only one functional (non-abstract) building block implementation included in the XCG framework. These blocks represent text snippets and are useful when generating source code. Whether the produced text gets stored in a text file or elsewhere depends on the output facility.

Please refer to the Javadoc for the implementation details. This is available on my homepage, too. In this paper I only want to drop a few words about the functional principles. The whole component is made up of the following classes:

Class	Comment
<code>BuildingBlock4Text</code>	Offers the public methods inherited from <code>BuildingBlock</code> to set variable values and to return preliminary output. Additionally, it includes internal methods used by <code>BuildingBlock4TextCollection</code> to load building blocks from the resource file.
<code>BuildingBlock4TextCollection</code>	Basically loads a set of text building blocks from a data-source. It uses <code>BuildingBlock4TextParser</code> to parse the resource file and create appropriate blocks.
<code>BuildingBlock4TextParser</code>	Parses the building block resource file and thereby establishes the format of that resource file. It's a XML format, which will be discussed later on.
<code>TemplateFragment</code>	A text building block consists of text fragments, which can be invariant or variable. This class serves as a base class for both fragment types.
<code>CharacterData</code>	Represents the invariant parts of a text building block.
<code>Variable</code>	Represents the variable part of text building blocks.

Table 3: `BuildingBlock4Text` classes

Just to be precise, even though the class `Variable` is listed under the class `BuildingBlock4Text`, it not only belongs to the text type building block described in this subsection. It's the general class representing variables and gets created by the abstract base class `BuildingBlock` any time a new variable gets declared in the resource file.

6.4.1 Format of resource file for text building blocks

The root tag is `<building-blocks>`. It includes multiple occurrences of `<building-block>` tags, meant to define a single building block. `<building-block>` has a compulsory attribute `name`, which is the unique name of the particular block. If the block has variables (not required), than the first immediate child has to be a `<variables>` tag containing multiple occurrences of `<variable>` tags. These define a variable. Therefore the only attribute `name` defines a name for that variable, which has to be unique within the current building block. If a variable declaration has content, this content will be taken as the variable's default values. Last but not least a block definition will contain another immediat child: `<template>`. This is mixed content, where the text part is invariant text and XML tags are interpreted as variable names. Before a block contributes to the output, these variables will be replaced by "real" values. Now we understand, that declaring variables offers the possibility to check that the user doesn't define variable names which might also have a meaning in the output. One could derive it's own building block class to generate HTML documents. In that case you need to ensure, that no variable can be named `
` or maybe `<p>`. Because then, you couldn't distinguish output from variables within the `<template>` section.

6.4.2 Example: producing output

The following example is meant to illustrate the in-memory representation of a text building block and its XML equivalent from a resource file or a database. Additionally it also shows how a text building block's `getPreliminaryOutput()` method produces output and replaces variables. This is a the XML representation (just a snippet):

```
<building-block name="beginClass">
  <variables>
    <variable name="name"/>
    <variable name="ancestor">Object</variable>
  </variables>
  <template>
    class <name/> extends <ancestor/>
  </template>
</building-block>
```

And this is the in-memory representation:

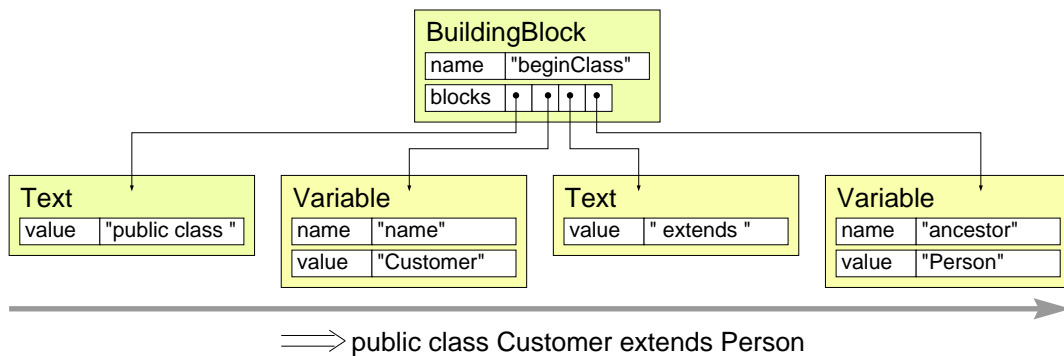


Figure 2: Building Block example

The output gets produced by "collecting" the values from the left to the right. Where text fragments are taken as they are, variables are replaced by their current values. Assuming `name = "Customer"` and `ancestor = "Person"` the following output will be produced: `"public class Customer extends Person"`.

Please note, that the variable with name "ancestor" has a default value "Object". If the generator would not set that variable's value explicitly, its default value would go to the outpt.

The next picture is meant to illustrate the internal representation of the variable list and the list of fragments, containing out of variable and invariant parts. The variable list is a feature introduced by `BuildingBlock`. `BuildingBlock4Text` introduces the list of fragments. This list refers to variables holding default and current values and it also refers to the invariant parts. References are done in order as found in the resource file. Output can be created by iterating over the fragment list from the left to the right and ask each fragment for it's value. In order to be able to do so, both, variable and invariant parts need to have the same interface, need to offer the same method `getValue()`. Therefore both classes are derived from `TemplateFragment`.

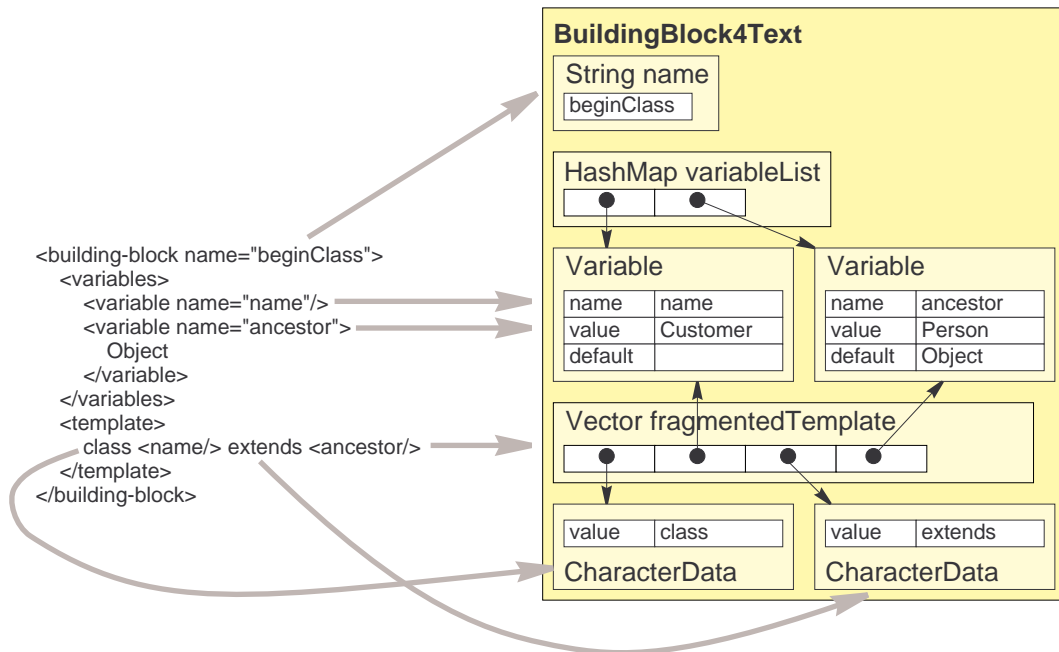


Figure 3: Internal representation of variable and invariant building block elements

7 Input

As said above there is no explicit input classes. The input is represented by the providers, which are also used by other components to access datasources.

8 Output

8.1 Class `OutputFacilityFactory`

Similar like with the other components, you can't directly create an output facility. Therefore you need to retrieve an instance of the output facility factory and need to ask that factory to create an instance of an output facility for you. The two methods below are serving that purpose.

```

public static OutputFacilityFactory getFactory( Configuration cfg )
throws XCGFOutputException;
public OutputFacility createInstance( String registryAlias,
String resourceDescriptor ) throws XCGFOutputException,
XCGFProviderException;

```

8.2 Class OutputFacility

To use that class, you need to know four methods:

- `open()` opens the data destination for write access.
- `getBuffer()` returns a buffer to which to write the preliminary output retrieved from the building block to.
- `write()` writes all buffers to the data destination.
- `close()` closes the data destination and frees up system resources.

The kind of the data destination is determined by the output facility at hand. To implement your own output facility, you need to implement the following methods:

```
protected abstract boolean resourceDescriptorNeeded();
protected abstract KeyValueTracker setUpParsingNetwork()
throws XCGFOutputException;
protected abstract void doSpecificOpen()
throws XCGFOutputException;
protected abstract Buffer createNewBuffer( String Name )
throws XCGFOutputException;
protected abstract void doSpecificWrite( Object output )
throws XCGFOutputException;
public abstract void doSpecificClose()
throws XCGFOutputException;
```

Most of the above methods are self-explanatory, others deserve to be explained (also refer to the Javadoc available on my homepage).

- `resourceDescriptorNeeded()` returns `true` in case the output facility requires a resource descriptor. Most facilities will, but in case it doesn't, return `false`.
- `setUpParsingNetwork()` is meant to set up the parser for parsing the resource descriptor.
- `createNewBuffer()` needs some more explanation. The idea behind the buffers is the following. Imagine you want to generate an HTML document, which starts off with a table of content (links) followed by the content. This would require to iterate over the input two times. First to generate the TOC and second to generate the content. This is where the buffers come into the game. Simply create two buffers. Make your generator to iterate over the input only once, but generate the TOC on the first buffer and the content on the second one. Now, when `write()` gets called, both buffers are getting written to the same file. First buffer one, second buffer number two—here we go.

8.3 Class Buffer

The method you have to call, when working with buffer objects is the one listed below. This method adds the (preliminary) output originating from the building block to the buffer. When `write()` gets called, the buffer content gets written to the data destination, may it be a file, a database or a queue.

```
public void addOutput( BuildingBlock output )
throws XCGFOutputException;
```

The following method needs to be implemented by you, when creating your own buffer classes. It makes sure, that the buffer accepts only a certain kind of building blocks. The method has to throw an exception, if the building block passed to it isn't of the right type. Otherwise it has to return normally.

```
public abstract void assertBuildingBlockClass( BuildingBlock output )
throws XCGFOutputException;
```

8.4 Class `OutputFacility4Text`

This output facility specifies on generating text. It is not specific to any data destination type yet. Use this class as a base class for generating text, which has to be stored in a file, in a database etc. Being specific for text output is achieved by implementing the abstract method `createNewBuffer()` in a way that it only generates buffers of type `Buffer4Text`. This type of buffers only accepts building blocks, which represent snippets of text: `BuildingBlock4Text`.

8.5 Class `OutputFacility4Text2File`

This facility is suited for generating text files. Its capability to handle text is inherited from `OutputFacility4Text`. Its specificness to data destinations of type file is achieved by implementing abstract methods as listed below.

Registry alias:

`Text2File`

Resource descriptor:

```
<connect info> ::= <file>
<file> ::= 'file' '=' any string
```

This is, what the class is doing usually ;-)

- It's specific open method (`doSpecificOpen()`), opens a file and keeps it open.
- It's specialised write method (`doSpecificWrite()`) writes the output objects, taken over from the building blocks, to the file. When doing that, it assumes, that the output object is of type `String`, which is the case if the building block was of type `BuildingBlock4Text`.
- `doSpecificClose()` closes the file opened by `doSpecificOpen()`.
- `createNewBuffer()` creates buffers of the appropriate type: `Buffer4Text`.
- `resourceDescriptorNeeded()` of course returns `true` since a descriptor is needed to determine the file name for the destination file. No surprise, that `setUpParsingNetwork()` sets up the needed parsing framework.

8.6 Class `Buffer4Text`

All the functionality needed was already implemented in the base class `Buffer`. The only additional functionality lies in the method `assertBuildingBlockClass()`. If you try to add the wrong type of building block to the output, an exception of type `XCGFOutputException` will be thrown. The only blocks accepted by this buffer is `BuildingBlock4Text`.

9 Generator

There are a few basic generator classes provided by the framework. But you are also free to create your own generator independently of these classes.

9.1 Class `Generator`

Very little functionality is realised by this class. There are two constructors available. Both need the following parameters: input (provider alias and resource descriptor), building block type (alias), block resource (provider alias and descriptor), control file (descriptor) and an optional list of output facilities (alias and descriptor). The second constructor is meant to be directly called by a program and receive the command line parameters (`main(String[] args)`).

```
public Generator( String inputAlias, String inputDescriptor,
                 String blockAlias, String blockProviderAlias, String blockDescriptor,
                 HashMap outputAliasesAndDescriptors, String controlDescriptor )
```

```
throws XCGFGeneratorException, XCGFConfigurationException,  
        XCGFOutputException, XCGFBuildingBlockException,  
        XCGFProviderException;  
public Generator( String[] args );
```

As meant for usage in command line generator, the last constructor catches all exceptions, prints errors to the standard error output and exits the program by calling `System.exit()` with an appropriate exit code.

Why are all arguments mandatory but the parameters concerning output facilities are not? This is because in some cases, the name of the destination file for the generated code depends on the input. In that case the generate will create output facilities when needed. But you can also make output facility parameters compulsory.

```
protected abstract int getMaxNumberOfOutputFacilitiesOnCommandLine();  
protected abstract int getMinNumberOfOutputFacilitiesOnCommandLine();
```

Above methods allow you to determine the minimum and maximum numbers of output facilities needed to be passed as an argument to the constructors. If the minimum is zero, no parameters are required but might be allowed. If the maximum is zero, no parameters are allowed.

The program flow of the constructor is illustrated below:

1. Load the configuration (it is passed to the factories).
2. Get a provider factory.
3. Get a provider from the factory, which will serve as input.
4. Get a output facility factory.
5. Get a building block collection factory.
6. Ask the factory for a building block collection.
7. Open all given output facilities (ask the factory). Do nothing, if no parameters regarding output were passed to the program.
8. Call `openControl()` to open the control file.
9. Call `generate()` to generate code.
10. Call `close()` to close all resources.
 - (a) Call `closeControl()` to close the control file.
 - (b) Call `cleanUp()` to close other resourced and to free up system resources.

Any time an output facility gets created, be it due to parameters, passed to the constructor or be it by calling `getNewOutputFacility()`, that output facility gets stored in a list. You can access these facility by applying `getOutputFacility()`.

9.2 Class GeneratorSAX

In addition to the base generator, this class requests an `InputSource` from the input provider, assuming the input shall be processed by applying the SAX parsing API.

9.3 Class GeneratorDOM

In addition to the base generator, this class requests an `Document` from the input provider, assuming the input shall be processed by processing a DOM tree.

9.4 Class GenerateDOMPreOrder

This generator base class build upon `GeneratorDOM`. It walks the input DOM tree in pre order (depths first) and calls code generation methods for every node, before and after all children etc.

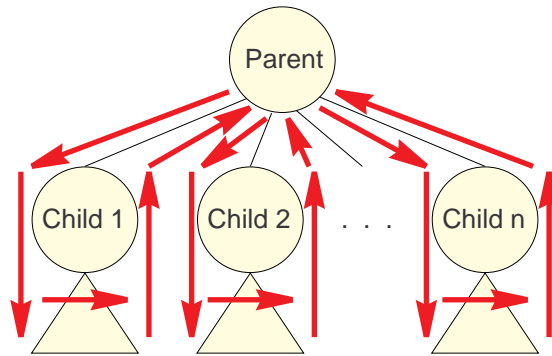


Figure 4: Walking the input DOM tree in pre-order

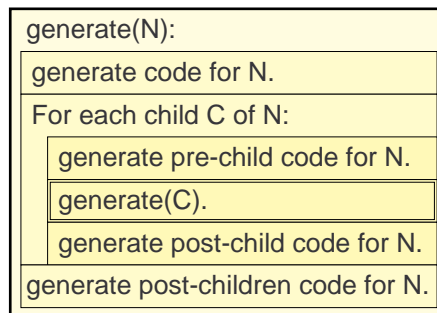


Figure 5: Code generation methods called, when walking the tree

10 Example: a poor man's XSLT

The following is meant to clarify the things said before. There is an example code generator included with this download⁶. This generator will serve as an example on what is going on with XCGF's data structures but it also demonstrates how to build a code generator using this framework.

Purpose of the sample code generator is to generate simple Java classes, which just contain properties (private attributes with public set and get methods for each attribute). Therefore the input has to provide the generator with the following information: class name and list of properties for that class (attribute name and type).

Since this generator features providers, there are various ways the input might be encoded as. We might like to use a XML file or we might want to model the classes to be generated in a database. Both ways will be demonstrated here.

10.1 Input coming from a XML file

Lets start with the XML file. I have chosen this file to look like this:

```
<classes>
  <class>
    <name>Person</name>
    <attributes>
      <attribute>
        <name>FirstName</name>
        <type>String</type>
      </attribute>
      <attribute>
        <name>LastName</name>
        <type>String</type>
      </attribute>
    </attributes>
  </class>
</classes>
```

10.2 Input coming from a database

A database representation might look like this.

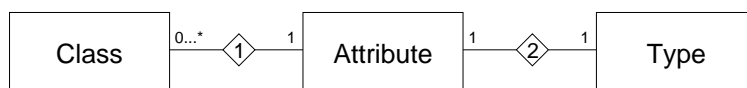


Figure 6: Entity Relationship Diagram

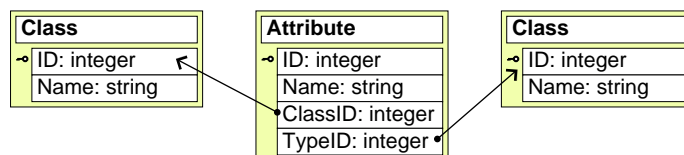


Figure 7: Physical data model

⁶Please refer to the Javadoc available on my homepage for more details.

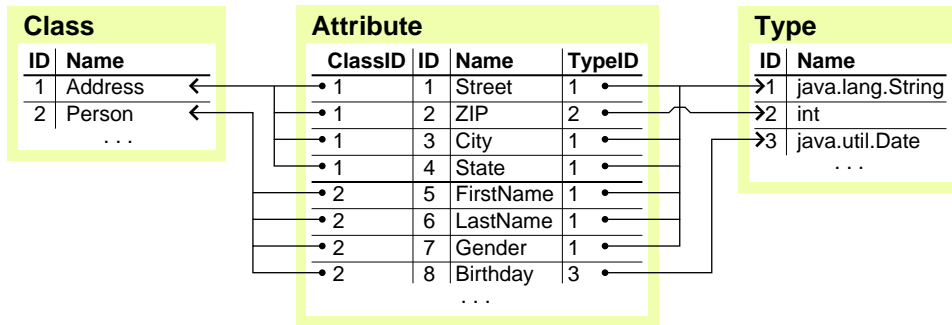


Figure 8: Example data

The JDBC provider presents that database content to the generator as an XML document, which looks like this:

```

<class-list>
  <class>
    <name>address</name>
    <attribute-list>
      <attribute>
        <name>Street</name>
        <type-list>
          <type>
            <name>java.lang.String</name>
          </type>
        </type-list>
      </attribute>
      <attribute>
        <name>ZIP</name>
        <type-list>
          <type>
            <name>int</name>
          </type>
        </type-list>
      </attribute>
      <attribute>
        <name>City</name>
        <type-list>
          <type>
            <name>java.lang.String</name>
          </type>
        </type-list>
      </attribute>
      <attribute>
        <name>State</name>
        <type-list>
          <type>
            <name>java.lang.String</name>
          </type>
        </type-list>
      </attribute>
    </attribute-list>
  </class>
</class>

```

```

<name>person</name>
<attribute-list>
  <attribute>
    <name>FirstName</name>
    <type-list>
      <type>
        <name>java.lang.String</name>
      </type>
    </type-list>
  </attribute>
  <attribute>
    <name>LastName</name>
    <type-list>
      <type>
        <name>java.lang.String</name>
      </type>
    </type-list>
  </attribute>
  <attribute>
    <name>Gender</name>
    <type-list>
      <type>
        <name>java.lang.String</name>
      </type>
    </type-list>
  </attribute>
  <attribute>
    <name>Birthday</name>
    <type-list>
      <type>
        <name>java.util.Date</name>
      </type>
    </type-list>
  </attribute>
</attribute-list>
<class>
</class-list>

```

10.3 Sample building block collection

The following building block collection is suited to generate (simple) Java classes.

```

<building-blocks>

  <building-block name="beginClass">
    <variables>
      <variable name="name"/>
    </variables>
    <template>
      public class <name/>
      {
    </template>
  </building-block>

  <building-block name="endClass">
    <template>
      }
    </template>

```

```

</building-block>

<building-block name="property">
  <variables>
    <variable name="name"/>
    <variable name="type"/>
  </variables>
  <template>
    private <type/> <name/>;
    public void set<name/>( <type/> value )
    {
        this.<name/> = value;
    }
    public <type/> get<name/>()
    {
        return <name/>;
    }
  </template>
</building-block>

</building-blocks>

```

10.4 Control language

Now we need to design a code generator, which is capable of generating Java classes from both data-sources introduced before. Therefore the generator is implemented as a simple XSLT processor—at least the poor man’s version of XSLT.

The generator defines a simple language for the control script. It reads that script and executes it, thereby transforming input into output by applying the templates given above.

```

<control script> ::= "if" <match> ":" {<statement>"}1..*
<match> ::= an XML tag from the input
<statement> ::= <open> | <print> | <apply>
<open> ::= "open" <tag> <extension>
<print> ::= "print" <template> {<tag>}0..*
<apply> ::= "apply"

```

The if statement states, that the following actions will be performed for every input element (node), which matches the given tag name <matc>.

10.4.1 Statement "open"

The generator is designed to produce text files only. No other storage media is supported. Even though it can be used for generating other files than Java source files, the ability to generate Java files requires to open output files based driven by data from the input. This is, because Java files must be named after the class they contain. Giving output facilities on the command line doesn’t make sense. The generator needs to process the input in order to decide which destination file to create.

This is exactly the task of the **open** command, where <tag> stands for any immediate child node⁷ of that node this action is assigned to by the <if> statement. <extension> can be any string constant. It will be attached to the file name. In case of generating Java you will use "java" as the extension, this will result in the file name <tag>.java.

The **open** statement acquires an implicit buffer object from the opened output facility. The **print** statement will write to that buffer. Think of extending the generator by another statement to explicitly acquire buffers and therefore gaining control over the location were to generate code within the output file!

⁷Please note: the given example generator doesn’t understand XPath expressions. In a "real" generator, you should allow to specify XPath expressions to address any input element, not only immediat children.

10.4.2 Statement "print"

The print command is used to produce output. The building block with name `template` will be used. The variables, if any declared, will be replaced by values from the input. These values are denoted by `<tag>`, which can be any immediate child of the node matched by the `if` statement.

10.4.3 Statement "apply"

This command suspends execution of the current statement block and the generator will go on processing all children of the current node. After all children have been processed, execution of this statement block will be resumed. This is important to generate "brackets" around other generated code.

10.4.4 Sample control script

To generate Java classes based on the XML input file given above, we can use the following control script:

```
if class :
    open name java;
    print beginClass name;
    apply;
    print endClass;

if attribute :
    print property type name;
```

For each `<class>` tag in the input file, the following actions will be performed:

- Open a text file, where the file's name is the value of the `<name>` tag, which is the immediate child of the matching `<class>` node. File extension is ".java".
- Put the building block to the output, where the name of that block is `beginClass` and its only variable will reflect the value of the same `<name>` tag mentioned already. This block will start the declaration of a java class.
- Now apply all child nodes of `<class>`. This will hit all `<attribute>` tags, which are children of this `<class>` node.
- After all attributes have been generated, close the class declaration by applying the building block `endClass`. This block doesn't have any variables, it's just a closing curled brace.

For each `<attribute>` tag in the input file, the following actions will be performed:

- Put the building block with name `property` to the output. This will define an property with a given name and type. These information can be taken from the immediate childs of `<attribute>`—`<name>` and `<type>`.

10.5 Running the example

To run the example you need to enter the following line at command line or have an equivalent setup done for your IDE:

```
java -cp xcgf_examples_1.1.jar;xcgf_1.1.jar;jutil.jar;jkvpf.jar;xercesImpl.jar;xmlParserAPIs.jar
    PoorMansXSLT
    -controlDescriptor ./examples/control.script
    -inputAlias XMLFile -inputDescriptor file=./examples/input.xml
    -blockAlias Text -blockProviderAlias XMLFile
    -blockDescriptor file=./examples/blocks.xml
```

11 Installation

Installation of XCGF is quite easy. You need to download the following Java archives and have them included in your classpath in order to be able to use XCGF: `jutil.jar`, `jkvpf.jar` and `xcgf_1.1.jar`. Additionally you need an XML parser, I'm using Xerces: `xercesImpl.jar` and `xmlParserAPIs.jar`. For running the unit test you need `junit.jar`. It is most convenient to use Ant as a build tool for compiling, jar-ing and generating Javadoc.

12 Unit test

This download also includes an unit test based on JUnit. Please refer to the Javadoc for the testing details.

A List of project files

File Name	Comments
<code>bin\xcgf_1.1.jar</code>	Java archive.
<code>...leiter\xcgf\buildingblocks\BuildingBlock.java</code>	Source, see Javadoc.
<code>...r\xcgf\buildingblocks\BuildingBlock4Text.java</code>	Source, see Javadoc.
<code>...\buildingblocks\BuildingBlock4TextParser.java</code>	Source, see Javadoc.
<code>...f\buildingblocks\BuildingBlockCollection.java</code>	Source, see Javadoc.
<code>...ldingblocks\BuildingBlockCollection4Text.java</code>	Source, see Javadoc.
<code>...ingblocks\BuildingBlockCollectionFactory.java</code>	Source, see Javadoc.
<code>...leiter\xcgf\buildingblocks\CharacterData.java</code>	Source, see Javadoc.
<code>...ter\xcgf\buildingblocks\TemplateFragment.java</code>	Source, see Javadoc.
<code>...\zahnleiter\xcgf\buildingblocks\Variable.java</code>	Source, see Javadoc.
<code>...org\zahnleiter\xcgf\config\Configuration.java</code>	Source, see Javadoc.
<code>...rg\zahnleiter\xcgf\exceptions\ErrorCodes.java</code>	Source, see Javadoc.
<code>...gf\exceptions\XCGFBuildingBlockException.java</code>	Source, see Javadoc.
<code>...gf\exceptions\XCGFConfigurationException.java</code>	Source, see Javadoc.
<code>...zahnleiter\xcgf\exceptions\XCGFException.java</code>	Source, see Javadoc.
<code>...ter\xcgf\exceptions\XCGFFactoryException.java</code>	Source, see Javadoc.
<code>...r\xcgf\exceptions\XCGFGeneratorException.java</code>	Source, see Javadoc.
<code>...iter\xcgf\exceptions\XCGFOutputException.java</code>	Source, see Javadoc.
<code>...er\xcgf\exceptions\XCGFProviderException.java</code>	Source, see Javadoc.
<code>src\org\zahnleiter\xcgf\generator\Generator.java</code>	Source, see Javadoc.
<code>...g\zahnleiter\xcgf\generator\GeneratorDOM.java</code>	Source, see Javadoc.
<code>...iter\xcgf\generator\GeneratorDOMPreOrder.java</code>	Source, see Javadoc.
<code>...g\zahnleiter\xcgf\generator\GeneratorSAX.java</code>	Source, see Javadoc.
<code>src\org\zahnleiter\xcgf\output\Buffer.java</code>	Source, see Javadoc.
<code>src\org\zahnleiter\xcgf\output\Buffer4Text.java</code>	Source, see Javadoc.
<code>...rg\zahnleiter\xcgf\output\OutputFacility.java</code>	Source, see Javadoc.
<code>...hnleiter\xcgf\output\OutputFacility4Text.java</code>	Source, see Javadoc.
<code>...ter\xcgf\output\OutputFacility4Text2File.java</code>	Source, see Javadoc.
<code>...leiter\xcgf\output\OutputFacilityFactory.java</code>	Source, see Javadoc.
<code>src\org\zahnleiter\xcgf\providers\Provider.java</code>	Source, see Javadoc.
<code>...\zahnleiter\xcgf\providers\Provider4JDBC.java</code>	Source, see Javadoc.
<code>...nleiter\xcgf\providers\Provider4XMLFiles.java</code>	Source, see Javadoc.
<code>...ahnleiter\xcgf\providers\ProviderFactory.java</code>	Source, see Javadoc.
<code>...nleiter\xcgf\providers>SelectValueMapper.java</code>	Source, see Javadoc.
<code>src\org\zahnleiter\xcgf\util\DOMWriter.java</code>	Source, see Javadoc.
<code>web\xcgf_examples_1.1.jar</code>	Example

Table 4: Project files

B Related web pages

- [W1] **JDBC(TM) Database Access**
java.sun.com/docs/books/tutorial/jdbc/basics/index.html
JDBC tutorial by Sun Microsystems.
- [W2] **New Features in the JDBC 2.0 API**
java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/index.html
Java JDBC 2.0 tutorial by Sun Microsystems.
- [W3] **Xerces2 Java Parser Readme**
xml.apache.org/xerces2-j/index.html
Xerces 2 XML parser for java by Apache.org.

Disclaimer

The use of my page's content (programs, wiring diagrams, pictures, documents) is free for non-commercial purposes only.

The information in this document has been carefully reviewed and is believed to be reliable, but I do not assume any liability arising out of the application or use of any documents, programs or circuit described herein.

Furthermore I want to declare that I'm not responsible in any way for the content of other web pages, books and other sources I'm referring to.