

BioUML workbench

Contents

1. Introduction	4
1.1. Architecture overview	4
2. Installation and configuration	7
2.1. Requirements	7
2.2. Compile and run	7
3. Extension points	8
3.1. diagramExport	9
3.2. diagramImport	11
3.3. diagramTypeConverter	13
3.4. moduleType	14
3.5. diagramViewPart	16
3.6. diagramEditorPart	17
3.7. visiblePlugin	18
3.8. menuItem	19
3.9. wizardPage	20
3.10. repositoryActionsProvider	21
3.11. aboutDialog	22
3.12. lookAndFeel	23
3.13. function	24
3.14. hostObject	28
3.15. genehub	31
3.16. annotation	32
3.17. solver	33
3.18. method	34
3.19. helpSet	35
3.20. service	36
3.21. servlet	36
4. Data collections	38
4.1. Introduction	38
4.2. DataElement	38
4.3. DataCollection	41
4.4. DataCollectionInfo	42
4.5. Usage of Data Collections	43
4.6. Mutable and immutable Data Collections	45
4.7. CollectionFactory	48

4.8. Basic Data Collections	49
4.9. Filtering Data Collections	53
4.10. QuerySystem	55
5. Plug-in development	59
5.1. Create new plug-in	59
5.2. How to create SQL transformer	60
6. JavaScript	63
6.1. Functions and host objects	63
6.2. R JavaScript host object	63
7. Examples	65
7.1. SQL database	65
7.1.1. Overview	65
7.1.2. SQL database config file	66
7.1.3. SQL transformers	68

1 Introduction

1.1 Architecture overview

BioUML workbench is a plugin-based application framework that provides its extensibility and possibility of seamless integration of other tools for systems biology. It consists from an Eclipse platform (<http://www.eclipse.org>) runtime kernel that supports 'plug-ins' and a set of plug-ins that support database access, diagram editing, and biological systems simulation.

Plug-in based architecture

- Plug-in - is the smallest unit of BioUML workbench function that can be developed and delivered separately into BioUML workbench. Plug-ins are coded in Java. A typical plug-in consists of Java code in a JAR library, some read-only files, and other resources such as images, message catalogs, native code libraries, etc. A plug-in is described in an XML manifest file, called plugin.xml. The parsed contents of plug-in manifest files are made available programmatically through a plug-in registry API provided by Eclipse runtime.
- Extension points are well-defined function points in the system where other plug-ins can contribute functionality.
- Extension is a specific contribution to an extension point. Plug-ins can define their own extension points, so that other plug-ins can integrate tightly with them.

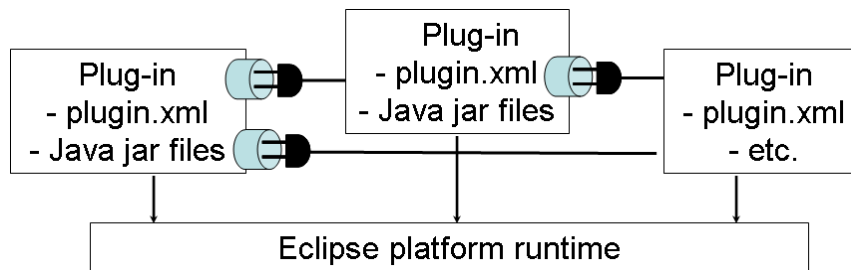



Figure 1.13. Plug-in based architecture,  - extension point;  - extension

Architecture of BioUML workbench

BioUML workbench installation includes a plugins folder where individual plug-ins are deployed. Each plug-in is installed in its own folder under the plugins folder. A plug-in is described in an XML manifest file, called plugin.xml, residing in the plug-in's folder. The parsed contents of plug-in manifest files are made available programmatically through a plug-in registry API provided by Eclipse runtime.

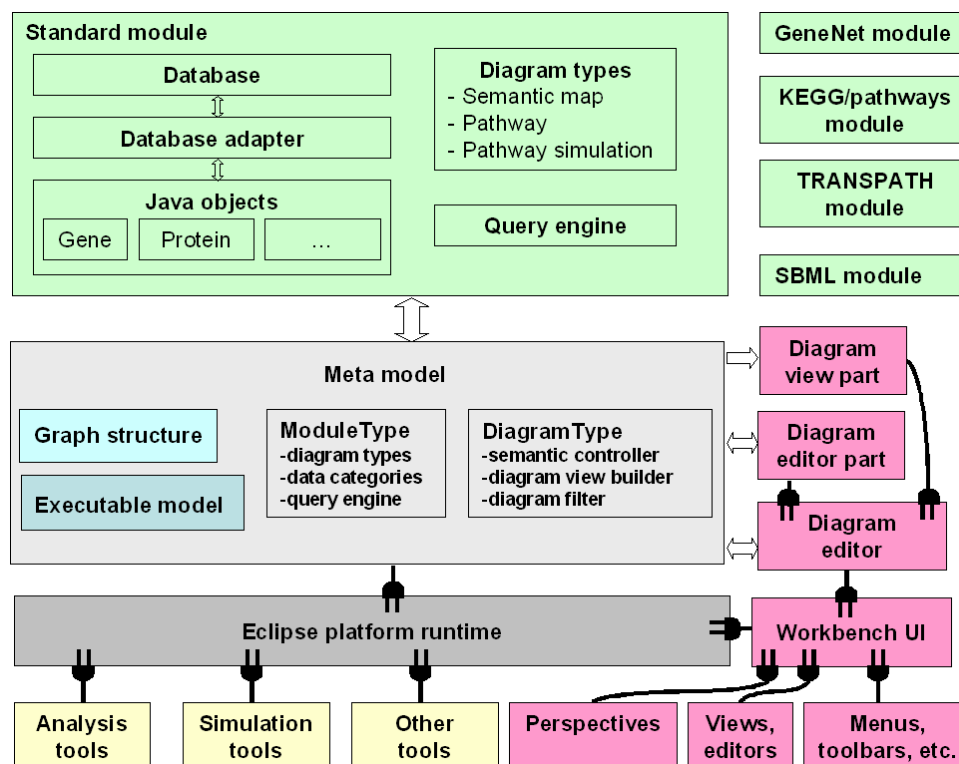


Figure 1.14. BioUML workbench - architecture overview.

Formal description and modeling of biological systems require coordinated efforts of different group of researchers:

- " programmers - they should provide computer tools for this task;
- " problem domain experts - they should specify what and how should be described;
- " experimenters and annotators - they should describe corresponding data following to these rules;
- " mathematicians - they should provide methods for models analysis and simulations.

BioUML architecture separates these tasks so they can be effectively solved by corresponding group of researchers and provides simple contract how these groups and corresponding software parts should communicate.

Architecture of BioUML server

BioUML server is Java application that is started as servlet on J2EE compatible server (we are using Tomcat server).

Like BioUML workbench it is also uses Eclipse runtime to manage by plug-ins that provide different services (Figure 1.15). Main services provided by BioUML server are:

- database service - provides information about database and secure access to it
- access service - provides access to databases (read/write)
- diagram service - provides protocol to read/write diagram and all diagram elements during one HTTP request
- Lucene service - provides full text search and its configuration
- query service - provides indexed search for a database

BioUML server supports access to different types of databases. Main of them are:

- relational databases (for example, Ensembl database that is available as MySQL dump)

- text databases (for example KEGG/Ligand database)
- XML databases (for example databases in BioPAX or SBML formats)
- databases available via web services (for example SABIO-RK database)

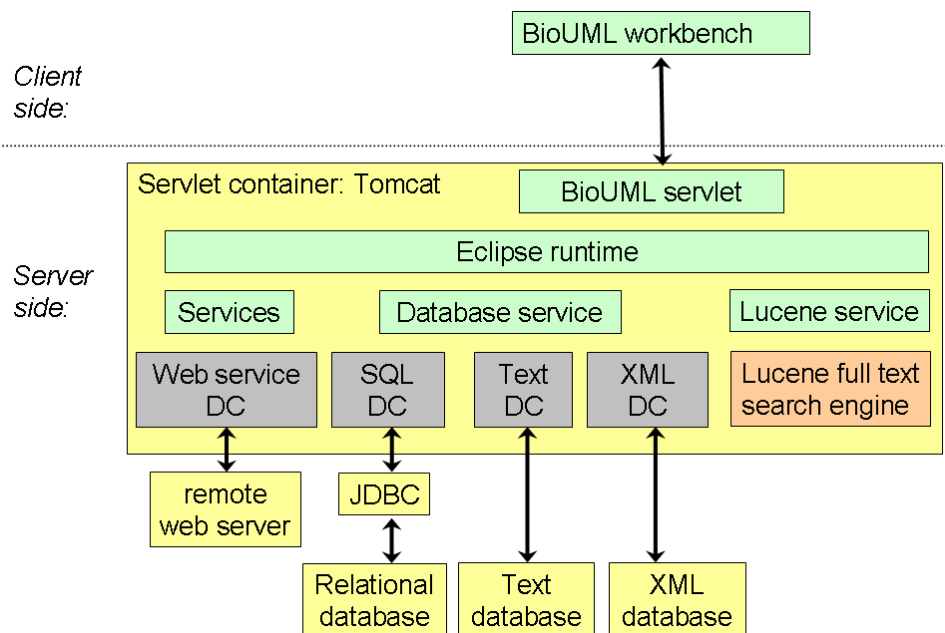


Figure 1.15. Architecture of BioUML server.

2 Installation and configuration

2.1 Requirements

Hardware

- Pentium II 500 or higher.
- 512 MB RAM.
- 50 MB free hard disk drive space.

Software

- Java virtual machine - JDK 1.6

You may download JDK 6 from Sun Microsystems Inc web site

<http://java.sun.com/javase/downloads/?intcmp=1281>

- Apache Ant

You may download Ant from <http://ant.apache.org/>

2.2 Compile and run

Download BioUML development kit (bdk.zip) and unpack to file system (for example, C:\Projects\java\BioUML).

Compile

The standard way of BioUML compilation is Ant target using. Go to <http://ant.apache.org/> for more information about Ant.

To compile BioUML project use ant target "plugin.all" (go to bdk\src folder and type "ant plugin.all" in command line). After successfully build you will see BUILD SUCCESSFUL string.

You can compile plug-ins separately. Each plug-in has its own Ant target. Use "usage" target (type "ant usage" in command line) to view full target list.

Run

To run BioUML use ant target "run" or executable file run.bat (run.sh for Unix).

Testing

BioUML uses JUnit for testing. All tests are grouped in *_test packages. To compile all tests use "ctest" target. To run tests use "rtest" target.

3 Extension points

BioUML workbench is a plugin-based application framework. It consists from a core runtime kernel that supports 'plug-ins' and a set of plug-ins that support database access, diagram editing, simulation for different complex systems etc.

Functionality is added to BioUML workbench using a common extension model.

Extension points are well-defined function points in the system that can be extended by plug-ins. When a plug-in contributes an implementation for an extension point, we say that it adds an extension to the platform. Plug-ins can define their own extension points, so that other plug-ins can integrate tightly with them.

The extension mechanisms is the most basic means of adding function to the platform and other plug-ins.

Module and diagram type issues

- [diagramExport](#)
- [diagramImport](#)
- [diagramTypeConverter](#)
- [moduleType](#)

User interface

- [diagramViewPart](#)
- [diagramEditorPart](#)
- [visiblePlugin](#)
- [menuItem](#)
- [wizardPage](#)
- [repositoryActionsProvider](#)
- [aboutDialog](#)
- [lookAndFeel](#)

JavaScript

- [function](#)
- [hostObject](#)

Plugins extensions

- [genehub](#)
- [annotation](#)
- [solver](#)

- [method](#)

Help

- [helpSet](#)

Server extensions

- [service](#)
- [servlet](#)

3.1 diagramExport

Identifier: biouml.workbench.diagramExport

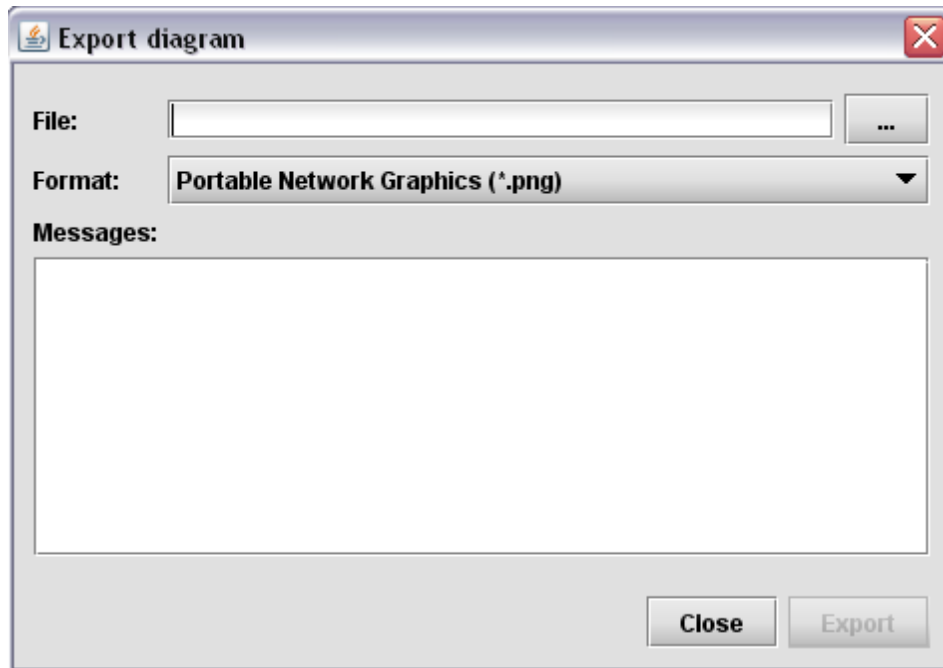
Since: 0.7.5

Description: using this extension point plug-in can provide diagram exports in different formats.

Format combo box in **Export diagram** dialog shows all formats suitable for selected diagram. This dialog is opened when user selects **Export diagram** menu item from **Diagram** section in main menu or from context dependent menu in repository.

List of suitable formats depends from diagram type, for example semantic network can not be exported in SBML format. DiagramExportRegistry is used to select suitable formats for the specified diagram type.

Conversion of diagram type to other type can be essential for export in some format. For example standard BioUML metabolic pathway diagram should be converted in SBML diagram type before export in SBML format.



Configuration Markup:

```
<!ELEMENT diagramExport>
<!ATTLIST diagramExport
    diagramType      CDATA      #REQUIRED
    format           CDATA      #REQUIRED
    exporter         CDATA      #REQUIRED
    suffix           CDATA      #IMPLIED
    convertTo        CDATA
    convertor        CDATA
    description      CDATA
>
```

- **diagramType** - class name of diagram type that can be exported in this format.
"*" means that this format, for example PNG, is suitable for all diagram types.
If some format is suitable for several diagram types, then diagram import section should be written separately for each diagram type.
- **format** - display name of format, for example "PNG".
- **exporter** - the fully-qualified name of a class which implements `biouml.model.DiagramExporter` interface.
- **suffix** - file suffix (extension) for the specified format.
- **convertTo** - the fully-qualified name of a class for diagram type to which diagram should be converted before export.
- **convertor** - the fully-qualified name of a class which implements `biouml.model.DiagramTypeConvertor` interface for diagram type conversion before export.
- **description** - format description. It can be plain or html text.

Examples:

Following example demonstrates definition of export of all diagram types in PNG format:

```
<extension    point="biouml.workbench.diagramExport">
    <export
        diagramType="*"
        format="PNG"
        suffix="png"
        class="biouml.workbench.diagram.PngDiagramExporter"
        description="%PNG_export"
    />
</extension>
```

API Information: The value of the exporter attribute must represent an implementer of `biouml.model.DiagramExporter` interface.

Pending: some formats support depends from JVM version, for example BMP and WBMP formats are supported only in Java 1.5. Additionally, the same exporter can support several formats. For this purpose `DiagramExporter` has special method, that returns true if the exporter was initialised successfully for the specified format and false otherwise:

```
boolean init(String format, String suffix)
```

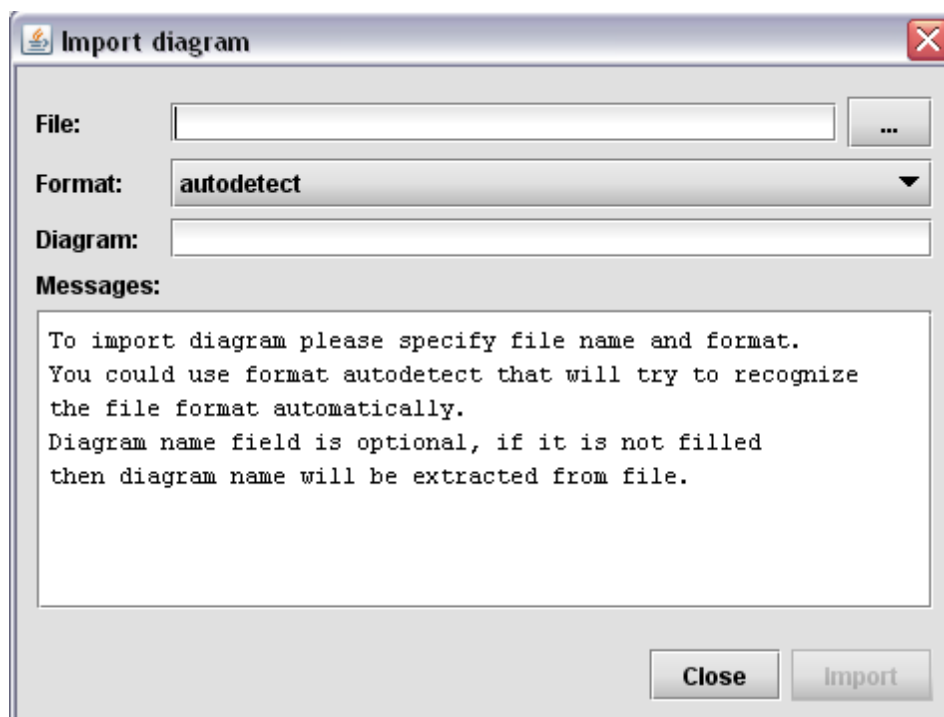
3.2 diagramImport

Identifier: `biouml.workbench.diagramImport`

Since: 0.7.5

Description: using this extension point plug-in can provide import of diagrams in different formats from external files.

Supported import formats will be listed in **Format** combo box in **Import diagram** dialog. This dialog is opened when user selects **Import diagram** menu item from **Diagram** section in main menu or from context dependent menu in repository.



Configuration Markup:

```
<!ELEMENT  import>
  <!ATTLIST  import
    format          CDATA      #REQUIRED
    importer        CDATA      #REQUIRED
    description     CDATA
  >
```

- **format** - display name of external format, for example "SBML level 1 version 1"
- **importer** - the fully-qualified name of a class which implements `biouml.model.DiagramImporter` interface.
- **description** - format description. It can be plain or html text.

Examples:

Following example demonstrates how SBML plug-in contributes support for import models in SBML level 1 version 1 format:

```
<extension  point="biouml.workbench.diagramImport">
  <import
    format="SBML level 1 version 1"
    class="biouml.plugins.sbml.SbmlImporter_11"
    description="%SbmlImporter_11_description"/>
```

</extension>

API Information: The value of the importer attribute must represent an implementer of `biouml.model.DiagramImporter` interface.

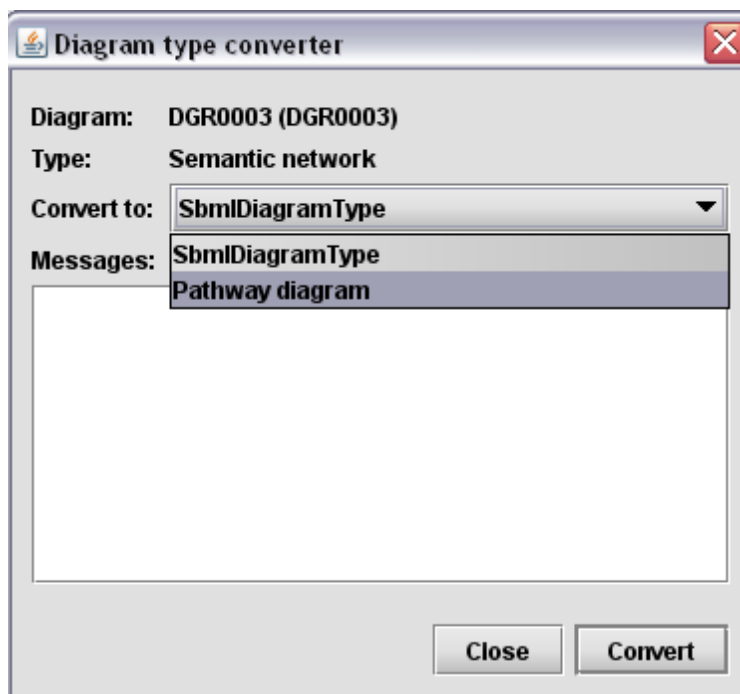
3.3 diagramTypeConverter

Identifier: `biouml.workbench.diagramTypeConverter`

Since: 0.7.4

Description: using this extension point plug-in can provide conversion of diagram from one diagram type to another.

Convert to combo box in **Diagram type converter** dialog shows all possible conversions for the selected diagram. This dialog is opened when user selects **Convert diagram** menu item from **Diagram** section in main menu or from context dependent menu in repository.



Configuration Markup:

```
<!--ELEMENT  conversion>
  <!--ATTLIST conversion
    diagramType      CDATA      #REQUIRED
    format            CDATA      #REQUIRED
    exporter          CDATA      #REQUIRED
    suffix            CDATA      #IMPLIED
```

```

        convertTo          CDATA
        convertor          CDATA
        description        CDATA
    >

```

- **diagramType** - class name of diagram type that can be exported in this format.
 "*" means that this format, for example PNG, is suitable for all diagram types.
 If some format is suitable for several diagram types, then diagram import section should be written separately for each diagram type.
- **format** - display name of format, for example "PNG".
- **exporter** - the fully-qualified name of a class which implements `biouml.model.DiagramExporter` interface.
- **suffix** - file suffix (extension) for the specified format.
- **convertTo** - the fully-qualified name of a class for diagram type to which diagram should be converted before export.
- **convertor** - the fully-qualified name of a class which implements `biouml.model.DiagramTypeConvertor` interface for diagram type conversion before export.
- **description** - format description. It can be plain or html text.

Examples:

The example demonstrates two possible conversions for `GeneNetworkDiagramType`:

```

<extension    id="standardDiagramTypeConversions"    point="biouml.workbench.dia
    <conversion
        from="biouml.standard.diagram.GeneNetworkDiagramType"
        to="biouml.standard.diagram.PathwayDiagramType"
        converter="biouml.model.DiagramTypeConverter$YesConverter"
    />
    <conversion
        from="biouml.standard.diagram.GeneNetworkDiagramType"
        to="biouml.standard.diagram.MetabolicPathwayDiagramType"
        converter="biouml.model.DiagramTypeConverter$ConditionalConverter"
    />
</extension>

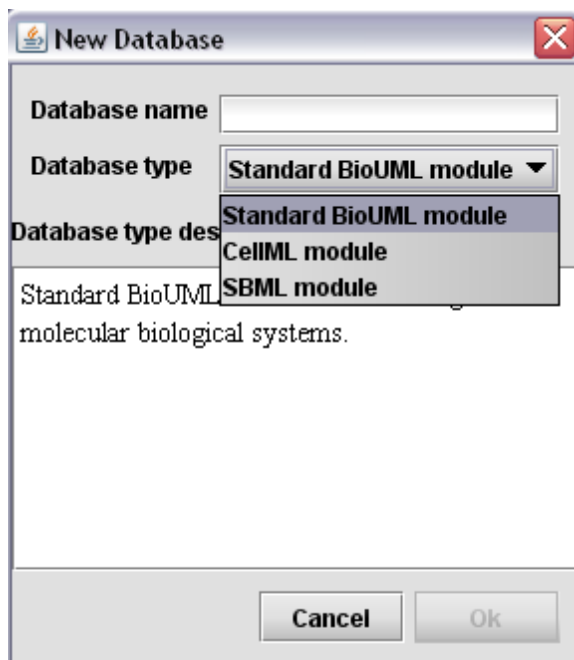
```

3.4 moduleType

Identifier: `biouml.workbench.moduleType`

Description: allows plug-in to contribute new module type that user can create using New Database dialog.

Available module types will be shown in **Database type** combo box in **New Database** dialog when user will select **Database > New simple database** menu item.



Configuration Markup:

```
<!ELEMENT moduleType>
  <!ATTLIST moduleType
    class          CDATA      #REQUIRED
    displayName    CDATA      #REQUIRED
    description    CDATA
  >
```

- **class** - the fully-qualified name of a class which implements `biouml.model.ModuleType`.
- **displayName** - a module type name as it will be shown in 'New module' dialog.
- **description** - a module type description as it will be shown in 'New module' dialog. It can be plain or html text.

Examples:

Following is an example of standard module type definition:

```
<extension point="biouml.workbench.moduleType">
  <moduleType
    class="biouml.standard.StandardModuleType"
```

```

        displayName="%standardModule"
        description="%standardModuleDescription"/>
</extension>

```

API Information: The value of the class attribute must represent an implementer of `biouml.model.ModuleType`. Additionally:

- method *canCreateEmptyModule()* should return true;
- module type class should implement method *createModule(Repository parent, String name)* should create new instance of corresponding module.

3.5 diagramViewPart

Identifier: `biouml.workbench.diagramViewPart`

Description: BioUML workbench provides special pane where viewers for some diagram data will be shown. Any plug-in can contribute specific viewers to show some diagram data. Typical example of viewer is Clipboard showing the copied element of diagram.

Configuration Markup:

```

<!ELEMENT  diagramViewPart>
  <!ATTLIST diagramViewPart>
    class                CDATA      #REQUIRED
    Name                 CDATA
    ShortDescription     CDATA
    Priority              CDATA
  >

```

- **class** - the fully-qualified name of a class which implements `ru.biosoft.gui.EditorPart`.
- **Name** - tab display name
- **ShortDescription** - editor part short description, can be used as tooltip
- **Priority** - tab priority, tabs will be sorted according they priority.

ViewPart attributes are mapped to corresponding Action properties using the associated key, so other Action or user defined attributes can be used.

See *Action.getValue(String key)* for details.

Examples:

Following is an example of a Clipboard definition:

```
<extension id="Clipboard" point="biouml.workbench.diagramViewPart">
  <diagramViewPart
    class="biouml.workbench.diagram.ClipboardView"
    Name="Clipboard"
    ShortDescription="Diagram clipboard is used to copy and paste diag
    Priority = "2.5"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of `ru.biosoft.gui.ViewPart`.

3.6 diagramEditorPart

Identifier: `biouml.workbench.diagramEditorPart`

Description: BioUML workbench provides special pane where editors for some diagram data will be shown. Any plug-in can contribute specific editors to modify some diagram data. Typical example of diagram editor is `DiagramDescriptionEditor` allowing user to edit diagram description as HTML or plain text.

Configuration Markup:

```
<!ELEMENT diagramViewPart>
  <!-- ATTLIST diagramViewPart -->
  class          CDATA      #REQUIRED
  Name           CDATA
  ShortDescription CDATA
  Priority        CDATA
-->
```

- **class** - the fully-qualified name of a class which implements `ru.biosoft.gui.EditorPart`.
- **Name** - tab display name
- **ShortDescription** - editor part short description, can be used as tool tip
- **Priority** - tab priority, tabs will be sorted according to their priority.

EditorPart attributes are mapped to corresponding Action properties using the associated key, so other Action or user defined attributes can be used.

See *Action.getValue(String key)* for details.

Examples:

Following is an example of a DiagramDescriptionEditor definition:

```
<extension id="Diagram description" point="biouml.workbench.diagramEditor"
  <diagramEditorPart
    class="biouml.workbench.diagram.DiagramDescriptionEditor"
    Name="Description"
    ShortDescription="Diagram description"
    Priority = "1.1"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of ru.biosoft.gui.EditorPart.

3.7 visiblePlugin

Identifier: ru.biosoft.workbench.visiblePlugin

Description: BioUML workbench provides special tab where 'visible' plug-ins will be shown. For this purpose any plug-in can provide one or more visible plug-in extensions.

There is special utility class ru.biosoft.plugins.VisiblePlugin from which other visible plug-ins can be inherited.

Configuration Markup:

```
<!ELEMENT visiblePlugin>
  <!ATTLIST visiblePlugin
    name          CDATA #REQUIRED
    class          CDATA #REQUIRED
    displayName    CDATA #IMPLIED
    description    CDATA
  >
```

- **name** - a unique name of plug-in.
- **class** - the fully-qualified name of a class which implements ru.biosoft.DataCollection.
- **displayName** - a plug-in title as it will be shown in plug-ins tab.
- **description** - a plug-in description.

Note: some other valid properties from ru.biosoft.DataCollection and its ancestor can be used if necessary.

Examples:

```
<extension point="ru.biosoft.workbench.visiblePlugin">
  <visiblePlugin
    name="SBW" class="biouml.plugins.sbw.SbwVisiblePlugin"
    displayName="Systems Biology Workbench"
    description="%sbw.descr"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of `ru.biosoft.plugins.VisiblePlugin`.

3.8 menuItem

Identifier: `biouml.workbench.menuItem`

Description: Any plugin can add items to main menu using `biouml.workbench.menuItem` extension point.

Configuration Markup:

```
<!ELEMENT menuItem>
  <!ATTLIST menuItem
    title          CDATA #REQUIRED
    parent         CDATA #REQUIRED
    action         CDATA #REQUIRED
  >
```

- **title** - the name of menu item.
- **parent** - top level menu name
- **action** - the fully-qualified name of action class for this menu item.

Examples:

Following is an example of a import BioPAX menu for BioPAX plugin:

```
<extension id="BioPAXImporter" point="biouml.workbench.menuItem">
  <menuItem
    title="Import BioPAX"
    parent="Database"
```

```

        action="biouml.plugins.biopax.imports.ImportBioPAXAction"
    />
</extension>

```

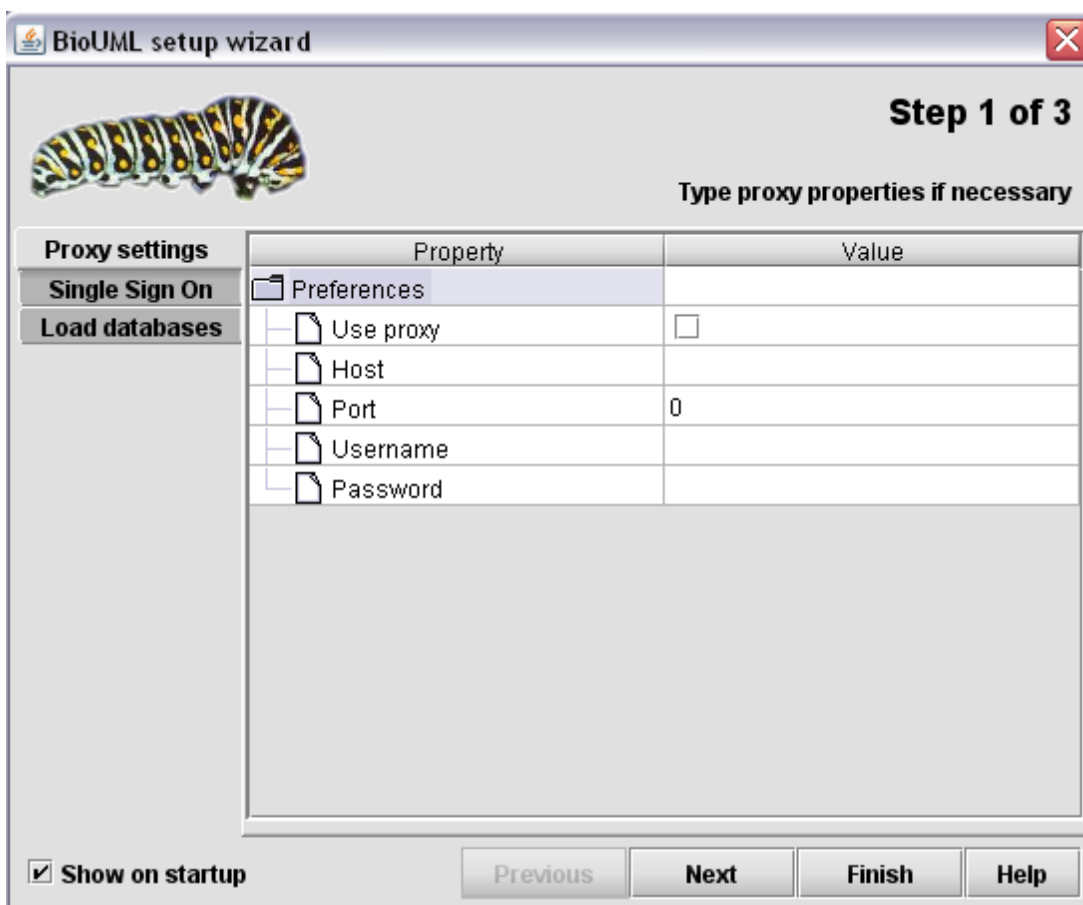
API Information: The value of the action attribute must represent an implementor of `javax.swing.Action`.

3.9 wizardPage

Identifier: `biouml.workbench.wizardPage`

Since: 0.8.5

Description: Define new step for BioUML setup wizard. Using this extension point plug-in can be configured from standard setup wizard.



Configuration Markup:

```

<!ELEMENT  wizardPage>
  <!ATTLIST wizardPage

```

```

    name          CDATA #REQUIRED
    description    CDATA #REQUIRED
    position       CDATA #REQUIRED
    page           CDATA #REQUIRED
  >

```

- **name** - the name of wizard step.
- **description** - short description of wizard step
- **position** - step number (beginning from 0).
- **page** - the fully-qualified name of wizard page class

Examples:

This is example of adding databases configuration page to setup wizard.

```

<extension id="Databases wizard page" point="biouml.workbench.wizardPage">
  <wizardPage
    name="Load databases"
    description="Load databases from server"
    position="2"
    page="biouml.workbench.module.DatabasesWizardPage"
  />
</extension>

```

API Information: The value of the page attribute must represent an implementor of `ru.biosoft.gui.setupwizard.WizardPage`.

3.10 repositoryActionsProvider

Identifier: `ru.biosoft.access.repositoryActionsProvider`

Description: When mouse is clicked under selected item in repository pane, then context sensitive pop up menu will be shown. Any plugin can contribute item specific actions into this pop up menu using `biouml.workbench.repositoryActionsProvider` extension point.

Configuration Markup:

```

<!ELEMENT repositoryActionsProvider>
  <!ATTLIST repositoryActionsProvider
    class          CDATA #REQUIRED
  >

```

- **class** - the fully-qualified name of a class which implements `ru.biosoft.access.repository.ActionsProvider`.

Examples:

Following is an example of a repository actions provider definition:

```
<extension id="repository actions" point="ru.biosoft.access.repositoryAct
    <repositoryActionsProvider class="biouml.workbench.RepositoryActions
</extension>
```

API Information: The value of the class attribute must represent an implementor of `ru.biosoft.access.repository.ActionsProvider`.

3.11 aboutDialog

Identifier: `ru.biosoft.workbench.aboutDialog`

Description: Makes a configuration of about dialog. The **id** of extension for main about dialog of BioUML must be `"aboutDialog"`.

Configuration Markup:

```
<!ELEMENT size>
<!ATTLIST size
    width          CDATA #REQUIRED
    height         CDATA #REQUIRED
>
```

- **width** - width of about dialog in pixels
- **height** - height of about dialog in pixels

Note: should be only one **size** element

```
<!ELEMENT tab>
<!ATTLIST tab
    name          CDATA #REQUIRED
    html          CDATA
    img           CDATA
```

>

- **name** - name of about tab
- **html** - path to HTML page for about tab
- **img** - path to image for about tab

Note: Use only one of **html** and **img** attributes in one tab.

Examples:

Following is an example of a BioUML about dialog

```
<extension id="aboutDialog" point="ru.biosoft.workbench.aboutDialog">
    <size width="400" height="200"/>
    <tab name="Info" img="biouml/workbench/resources/AboutLogo.png"/>
    <tab name="Web Pages" html="biouml/workbench/resources/about_web_page.html"/>
    <tab name="Contacts" html="biouml/workbench/resources/about_contacts.html"/>
    <tab name="Notes" html="biouml/workbench/resources/about_notes.html"/>
</extension>
```

3.12 lookAndFeel

Identifier: ru.biosoft.workbench.lookAndFeel

Since: 0.7.6

Description: using this extension point plug-in can contribute new look and feel.

This look and feel will be loaded both into UIManager and LookAndFeelRegistry.

Configuration Markup:

```
<!ELEMENT lookAndFeel (theme)*>
  <!-- ATTLIST lookAndFeel -->
  name CDATA #REQUIRED
  class CDATA #REQUIRED
  defaultTheme CDATA #IMPLIED
  >
```

- **name** - name of look and feel.
- **class** - the fully-qualified name of javax.swing.LookAndFeel subclass.
- **defaultTheme** - name of default theme.

```
<!ELEMENT theme>
  <!--ATTLIST theme
    name          CDATA      #REQUIRED
    class         CDATA      #REQUIRED
  -->
```

Examples:

This is example from plaf.metal plugin.

```
<extension id="plaf.metal.themes" point="ru.biosoft.workbench.LookAndFeel">
  <LookAndFeel name="Metal" defaultTheme="Steel">
    <theme name="Steel" class="javax.swing.plaf.metal.DefaultLookAndFeelMetalSteel"/>
    <theme name="Aqua" class="AquaTheme"/>
    <theme name="Charcoal" class="CharcoalTheme"/>
    <theme name="High contrast" class="ContrastTheme"/>
    <theme name="Emerald" class="EmeraldTheme"/>
    <theme name="Ruby" class="RubyTheme"/>
    <script name="getTheme"> LookAndFeel.getCurrentTheme(); </script>
    <script name="setTheme"> LookAndFeel.setCurrentTheme(theme); </script>
  </LookAndFeel>
</extension>
```

3.13 function

Identifier: ru.biosoft.plugins.javascript.function

Description: Using this extension plug-in can define JavaScript functions that will be available from JavaScript shell. Plug-in functions will be shown in Analyses/JavaScript/functions section in repository tree. Function help will be shown in View/Edit tab when the function item will be selected in repository tree.

When the JavaScript function is called the specified by extension class method will be invoked. The METHOD must match one of two forms: with fixed or variable number of arguments.

Fixed arguments number

The first form is a member with zero or more parameters of the following types: Object, String, boolean, Scriptable, byte, short, int, float, or double. The Long type is not supported because the double representation of a long (which is the EMCA-mandated storage type for Numbers) may lose precision. The return value must be void or one of the types allowed for parameters.

The runtime will perform appropriate conversions based upon the type of the parameter. A parameter type of Object specifies that no conversions are to be done. A parameter of type String will use Context.toString to convert arguments. Similarly, parameters of type double, boolean, and Scriptable will cause Context.toNumber, Context.toBoolean, and Context.toObject, respectively, to be called.

If the method is not static, the Java 'this' value will correspond to the JavaScript 'this' value. Any attempt to call the function with a 'this' value that is not of the right Java type will result in an error.

Variable arguments number

The second form is the variable arguments (or "varargs") form. The method must be a static Java method with parameters:

```
(Context cx, Scriptable thisObj, Object[] args, Function funObj)
```

args - parameter contains the arguments

thisObj - is the JavaScript 'this' value

funObj - is the function object for the invoked function.

The Java method should return an Object result.

Configuration Markup:

```
<!ELEMENT function (argument*, doc?)>
  <!-- ATTLIST function -->
    name          CDATA      #REQUIRED
    class          CDATA      #REQUIRED
    method         CDATA      #REQUIRED
    varargs        CDATA      "false"
    javadoc        CDATA
  >
```

- **argument** - function arguments.
- **doc** - the function description (documentation).
- **name** - the function name (how it will be used JavaScript).

- **class** - the fully-qualified name of a class that provides corresponding static method.
- **member** - the method or constructor that will be invoked to process the function call.
- **varargs** - indicates the function form: fixed or variable argument number.
- **javadoc** - URL to javadoc for the function description.

```
<!--ELEMENT argument>
  <!--ATTLIST argument>
    class          CDATA    #REQUIRED
  >
```

- **class** - the fully-qualified name of a argument type.

```
<!--ELEMENT doc (argument*, returns?, throws*, example*)>
  <!--ATTLIST doc>
    description    CDATA    #REQUIRED
  >
```

- **argument** - function argument description.
- **returns** - description of the function result.
- **throws** - description of exceptions that can be thrown by the function.
- **description** - function description.
- **example** - usage examples.

```
<!--ELEMENT argument>
  <!--ATTLIST argument>
    name          CDATA    #REQUIRED
    type          CDATA    #REQUIRED
    obligatory    CDATA    "true"
    description    CDATA
  >
```

- **name** - the argument name.
- **type** - the argument type.
- **obligatory** - indicates whether the argument is obligatory.
- **description** - the argument description.

```
<!--ELEMENT returns>
  <!--ATTLIST returns>
    type          CDATA    #REQUIRED
    description    CDATA
```

>

- **type** - the returned value type.
- **description** - the returned value description.

```
<!--ELEMENT throws>
  <!--ATTLIST throws>
    type          CDATA    #REQUIRED
    description   CDATA
  >
```

- **type** - the exception type.
- **description** - describes when and why the exception can be thrown.

```
<!--ELEMENT example>
  <!--ATTLIST example>
    code          CDATA    #REQUIRED
    description   CDATA
  >
```

- **code** - the code example.
- **description** - comment.

Examples:

This is example of function definition with variable number of arguments.

```
<extension id="defineClass" point="ru.biosoft.plugins.javascript.function"
  <function
    name="defineClass"
    class="ru.biosoft.plugins.javascript.Global"
    method="defineClass"
    varargs="true">
      <doc description="%defineClass.descr">
        <argument name="clazz" type="String" obligatory="true" description="...">
      <returns type="void"/>
      <throws type="IllegalAccessException"
        description="if access is not available to a reflected class" />
      <throws type="InstantiationException"
        description="if unable to instantiate the named class"/>
```

```

        <throws type="InvocationTargetException"
            description="if an exception is thrown during execution of me
        <throws type="ClassDefinitionException"
            description="if the format of the class causes this exception
        <throws type="PropertyException"
            description="if the format of the class causes this exception
    </doc>
</function>
</extension>

```

3.14 hostObject

Identifier: ru.biosoft.plugins.javascript.hostObject

Description: Using this extension point plug-in can provide access to particular Java objects (host objects) provided by plug-in. Plug-in host objects will be shown in 'Analyses/JavaScript/host objects' section in repository tree. Host object description (help) will be shown in View/Edit tab when the host object item will be selected in repository tree.

Configuration Markup:

```

<!ELEMENT hostObject (doc?)>
  <!ATTLIST hostObject>
    name          CDATA      #REQUIRED
    class         CDATA      #REQUIRED
    javadoc       CDATA
  >

```

- **name** - the function name (how it will be used JavaScript).
- **class** - the fully-qualified name of a class that provides corresponding static method.
- **javadoc** - URL to javadoc for the corresponding Java class.

```

<!ELEMENT doc (property*, function*)>
  <!ATTLIST doc>
    type          CDATA      #IMPLIED
    description   CDATA      #REQUIRED
  >

```

- **doc** - description of host object, its properties, functions and examples.

- **property** - description of host object property.
- **function** - description of host object function.
- **example** - host object usage example.

```
<!--ELEMENT property>
  <!--ATTLIST property>
    name          CDATA    #REQUIRED
    type          CDATA    #REQUIRED
    readOnly      CDATA    "false"
    description   CDATA    #REQUIRED
  >
```

- **name** - the property name.
- **type** - the property type.
- **obligatory** - indicates whether the property is read only.
- **description** - the property description.

```
<!--ELEMENT function (argument*, returns?, throws*, example*)>
  <!--ATTLIST function>
    name          CDATA    #REQUIRED
    description   CDATA    #REQUIRED
  >
```

- **argument** - function argument description.
- **returns** - description of the function result.
- **throws** - description of exceptions that can be thrown by the function.
- **example** - usage examples.
- **name** - function name.
- **description** - function description.

```
<!--ELEMENT argument>
  <!--ATTLIST argument>
    name          CDATA    #REQUIRED
    type          CDATA    #REQUIRED
    obligatory     CDATA    "true"
    description   CDATA
  >
```

- **name** - the argument name.
- **type** - the argument type.

- **obligatory** - indicates whether the argument is obligatory.
- **description** - the argument description.

```
<!--ELEMENT returns>
  <!--ATTLIST returns>
    type          CDATA    #REQUIRED
    description   CDATA
  >
```

- **type** - the returned value type.
- **description** - the returned value description.

```
<!--ELEMENT throws>
  <!--ATTLIST throws>
    type          CDATA    #REQUIRED
    description   CDATA
  >
```

- **type** - the exception type.
- **description** - describes when and why the exception can be thrown.

```
<!--ELEMENT example>
  <!--ATTLIST example>
    code          CDATA    #REQUIRED
    description   CDATA
  >
```

- **code** - the code example.
- **description** - comment.

Examples:

This is example from ru.biosoft.access.javascriptplugin description.

```
<extension id="dataFilter" point="ru.biosoft.plugins.javascript.hostObject"
  <hostObject name="dataFilter" class="ru.biosoft.access.javascript.JavaS
    <doc description="Facade for data-filtering">
      <function name="byValue" description="returns error from last
        <argument name="source" type="Object" obligatory="true" d
```

```

        <argument name="property" type="String" obligatory="true"
        <argument name="values" type="String" obligatory="true"
        <returns type="ru.biosoft.access.FilteredDataCollection" d
    </function>
    <function name="byExpression" description="filters by given j
        <argument name="source" type="Object" obligatory="true" d
        <argument name="expression" type="String" obligatory="true
        <returns type="ru.biosoft.access.FilteredDataCollection"
    </function>
    <function name="bySet" description="filters by given data coll
        <argument name="source" type="Object" obligatory="true" d
        <argument name="filterSource" type="Object" obligatory="t
        <returns type="ru.biosoft.access.FilteredDataCollection"
    </function>
</doc>
</hostObject>
</extension>

```

3.15 genehub

Identifier: biouml.plugins.microarray.genehub

Description: GeneHub provides a way to map **DatabaseReference** from diagram element references to microarray ids. You can add your own GeneHub for biouml. **plugins.microarray.plugin**.

Configuration Markup:

```

<!ELEMENT  hubItem>
  <!ATTLIST  hubItem
    name          CDATA #REQUIRED
    class         CDATA #REQUIRED
    host          CDATA #REQUIRED
    database      CDATA #REQUIRED
    port          CDATA #REQUIRED
    jdbc_driver   CDATA #REQUIRED
    user          CDATA #REQUIRED
    password      CDATA #REQUIRED
  >

```

- **name** - genehub name.
- **class** - the fully-qualified name of GeneHub class.

- **host** - database host
- **database** - database name
- **port** - database port
- **jdbc_driver** - database JDBC driver
- **user** - database user
- **password** - database password

Examples:

```
<extension id="GeneHubSupport" point="biouml.plugins.microarray.genehub">
  <hubItem
    name="Ensembl"
    class="biouml.plugins.genehub.GeneHubImpl"
    host="%host"
    database="%database"
    port="%port"
    jdbc_driver="%jdbc_driver"
    user="%user"
    password="%password"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of `biouml.plugins.microarray.GeneHub`.

3.16 annotation

Identifier: `biouml.plugins.sbml.annotation`

Description: SBML file may contains different information using **annotation** block. This extension point allows to register new **SbmlExtension** class which will be called when SBML file reading and writing. One **SbmlExtension** class should transform one structure to XML format. In our example, **ExperimentExtension** allows to save diagram experiments to SBML.

Configuration Markup:

```
<!ELEMENT annotation>
  <!-- ATTLIST annotation
    namespace          CDATA #REQUIRED
    priority            CDATA
    extension_class     CDATA #REQUIRED
```


>

- **namespace** - the name of XML element.
- **priority** - priority level for this extension (by default, 100).
- **extension_class** - the fully-qualified name of extension class.

Examples:

This is example of experiment extension for SBML format

```
<extension id="ExperimentAnnotationExtension" point="biouml.plugins.sbml.
  <annotation
    namespace="experiment"
    priority="3"
    extension_class="biouml.plugins.sbml.extensions.ExperimentExtension
  />
</extension>
```

API Information: The value of the `extension_class` attribute must represent an implementor of `biouml.plugins.sbml.extensions.SbmlExtension`.

3.17 solver

Identifier: `biouml.plugins.simulation.solver`

Description: This extension point allows to define new solver for model simulation.

Configuration Markup:

```
<!ELEMENT solver>
  <!-- ATTLIST solver
    class          CDATA #REQUIRED
    displayName    CDATA #REQUIRED
    type           CDATA #REQUIRED
  -->
```

- **class** - the fully-qualified name of solver class.
- **displayName** - the name of solver.
- **type** - solver type, may be "JAVA" or "MATLAB"

Examples:

This is example of DormandPrince solver definition.

```
<extension id="SolverType" point="biouml.plugins.simulation.solver">
  <solver
    class="biouml.plugins.simulation.ode.DormandPrince"
    displayName="DormandPrince"
    type="JAVA"
  />
</extension>
```

3.18 method

Identifier: ru.biosoft.analysis.method

Description: This extension point allows you to define new analyses method.

Configuration Markup:

```
<!ELEMENT analysisClass>
  <!ATTLIST analysisClass
    class          CDATA #REQUIRED
    group          CDATA #REQUIRED
    name           CDATA #REQUIRED
    description    CDATA #REQUIRED
  >
```

- **class** - the fully-qualified name of method class.
- **group** - name of group to define parent collection in repository.
- **name** - method name.
- **description** - path to the method description HTML file.

Examples:

Following is an example of HypergeometricAnalysis method definition.

```
<extension id="HypergeometricAnalysis" point="ru.biosoft.analysis.method">
  <analysisClass
    class="ru.biosoft.analysis.HypergeometricAnalysis"
```

```

        group="Statistics"
        name="Hypergeometric  analysis"
        description="ru/biosoft/analysis/resources/Hyper.html"
    />
</extension>

```

API Information: The value of the class attribute must represent an implementor of ru.biosoft.analysis.AnalysisMethod.

3.19 helpSet

Identifier: ru.biosoft.workbench.helpSet

Description: Help system of BioUML based on javax.help.HelpSet elements which creates automatically by .hs files. Use this extension point to define help for your plugin.

Configuration Markup:

```

<!ELEMENT  helpSet>
  <!ATTLIST helpSet
    id          CDATA #REQUIRED
    file        CDATA #REQUIRED
  >

```

- **id** - help set identifier.
- **file** - the name of .hs file

Examples:

This is example of BioUML workbench help

```

<extension id="workbench help" point="ru.biosoft.workbench.helpSet">
  <helpSet
    id="workbench help"
    file="workbench.hs"
  />
</extension>

```

3.20 service

Identifier: ru.biosoft.server.service

Description: **Service** is the way to communicate between BioUML client and BioUML server. Standard BioUML services are AccessService, QuerySystemService, DiagramService, ModuleService and LuceneService. Using this extension point plug-in can provide their own service to work with BioUML server.

Configuration Markup:

```
<!ELEMENT  service>
  <!ATTLIST  service
    name          CDATA  #REQUIRED
    class         CDATA  #REQUIRED
  >
```

- **name** - the name of service.
- **class** - the fully-qualified name of service class.

Examples:

Following is an example of access service definition:

```
<extension  id="access.service"  point="ru.biosoft.server.service">
  <service
    name="access.service"
    class="ru.biosoft.access.server.AccessService"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of ru.biosoft.server.Service.

3.21 servlet

Identifier: ru.biosoft.server.servlet

Description: BioUML server provides not only services for BioUML client connection. It also provides servlets for different applications. For example, BioUML server can return diagram images for other web-application. Using this extension point plug-in can provide their own servlets to access data from other applications.

Configuration Markup:

```
<!ELEMENT  servlet>
  <!ATTLIST  servlet
    class          CDATA #REQUIRED
    prefix         CDATA #REQUIRED
  >
```

- **class** - the fully-qualified name of servlet extension class.
- **prefix** - prefix for servlet address (address of servlet is **http://<biouml_host>/biouml/<prefix>**)

Examples:

Following is an example of a diagram servlet:

```
<extension  id="Ubiprot"  point="ru.biosoft.server.servlet">
  <servlet
    class="ru.biosoft.server.servlets.ubiprot.DiagramServlet"
    prefix="ubiprot"
  />
</extension>
```

API Information: The value of the class attribute must represent an implementor of `ru.biosoft.server.tomcat.ServletExtension`.

4 Data collections

Enter topic text here.

4.1 Introduction

Data Collections is a framework to facilitate processing of hierarchically organized data which cannot be readily presented in relational format. Main goals when introducing Data Collections were the following:

1. Provide mechanism for object mapping of information in biological databases.
2. Provide mechanism to package information in form compliant with JavaBeans™ technology for easier visualization
3. Provide a dynamic mechanism where new data formats can be easily “plugged in” later
4. Provide mechanism for querying the information from Data Collections using various criteria
5. Provide event notification mechanism for changes made in the data

There are four architectural elements of Data Collections framework.

1. DataElement – an atomic piece of information which Data Collections work with. DataElement can be thought as analogue of “record” or “relation” in the traditional relational database model
2. DataCollection – a group of DataElements of the same type (or the same “class” in terms of OOP model). DataCollection provides primitives for iterating through available DataElements and for adding/modifying/deleting DataElements inside it. When dealing with hierarchically organized data – DataCollections are becoming DataElements themselves.
3. CollectionFactory – main entry point for manipulating Data Collections. CollectionFactory is typically used for registering/deregistering Data Collections as well as for obtaining existing Data Collection via its complete name (or URL)
4. Repository – special DataCollection for storing hierarchical structure of DataCollections onto the local file system as well as onto some remote storage

Most of functionality of Data Collections framework resides in ru.biosoft.access package of the BioUML sources.

4.2 DataElement

Definition

DataElement is defined as follows in DataElement.java:

```
public interface DataElement
{
    String getName();
    DataCollection getOrigin();
}
```

From the definition of the interface it is clear that there are only two properties necessary for turning any Java object

into DataElement:

name—a unique name of DataElement in this DataCollection. This can be thought as similar to primary key value in relational databases

origin—reference to DataCollection which this DataElement belongs to. There are situations though where the same DataElement can be used in multiple Data Collections. In this case ‘origin’ property should refer to some main Data Collection where definition of is application-dependent.

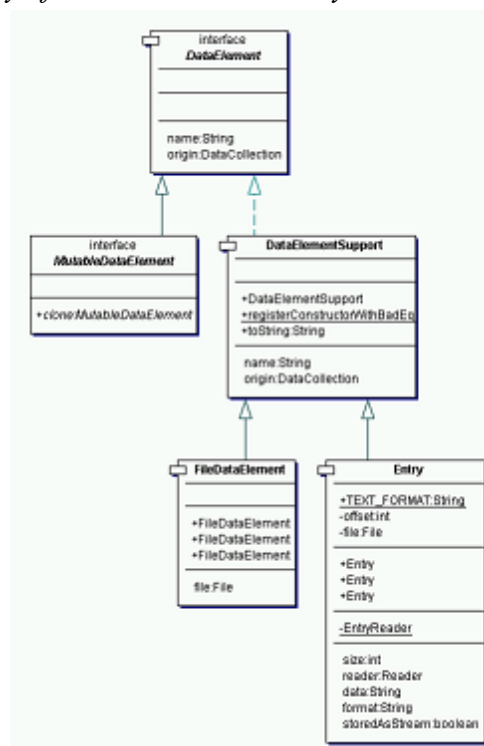
ru.biosoft.access package contains also the default implementation of DataElement – DataElementSupport located in DataElementSupport.java. Default implementation allows the developer to pass ‘name’ and ‘origin’ properties in a constructor when creating DataElement.

As mentioned, DataElement can be referenced from multiple DataCollections. In this case, there must be a mechanism to detach a DataElement from one DataCollection for putting it in into another. This mechanism is provided by using MutableDataElement interface which is defined as follows:

```
public interface MutableDataElement extends DataElement
{
    public MutableDataElement clone(DataCollection origin);
}
```

By invoking method clone on a MutableDataElement an application-dependent copy of the DataElement can be created and put into new DataCollection.

The diagram below shows hierarchy of DataElements used by DataCollection framework



Usages of DataElement

Entry – this is a common class for wrapping data entries from biological databases. This class is intended to represent information from flat files which can be information about sequence, matrix or individual site. Entry as DataElement is used by EntryCollection (see below).

Entry is defined as follows:

```
public class Entry extends DataElementSupport
{
    public Entry( DataCollection pParent, String pName, String pData, String pFormat );
    public Entry( DataCollection pParent, String pName, String pData );
    public Entry( DataCollection parent, String name, File pFile, long pOffset );

    public int getSize();
    public String getData();
    public java.io.Reader getReader();
    final public String getFormat();
    ...
}
```

The following properties of the Entry are defined

size – size of the entry's data in the file

data - entry's data presented in form of Java String

format – format in which entry is presented. Since Entry is introduced for dealing plain text files, the only valid value is 'text'.

reader – if this entry contains huge amount of data such as a long sequence, it is unreasonable to keep all entry's data entirely in the memory. Instead in such situation the data are kept on a disk and entry provides only Reader which reads data from a disk upon request.

FileDataElement - DataElement which wraps the java.io.File object. This class is used for storing of objects that represent local files. FileDataElement as DataElement is used by FileCollection (see below)

FileDataElement is defined as follows:

```
public class FileDataElement extends DataElementSupport
{
    /** File stored in this FileDataElement */
    protected File file;

    public FileDataElement( String name, DataCollection origin, File parent )
```



```

    public FileDataElement(String name, DataCollection origin, String parent) {
    public FileDataElement(String name, FileCollection origin);

    public File getFile();
    public void setFile(File file) throws Exception;
}

```

FileDataElement defines additionally the property 'file' which stores to the actual File object.

4.3 DataCollection

DataCollection interface is a core notion of Data Collections framework. It defines main methods for processing series of data. Data Collection can be thought as similar to a table in traditional semantics of relational databases.

At the same time DataCollection can be thought as a generic form of java.util.Collection, where developers may use traditional methodology for working with Collection with any data.

DataCollection is defined as follows:

```

public interface DataCollection extends DataElement
{
    int getSize();
    Class getDataElementType();
    boolean isMutable();
    DataCollectionInfo getInfo();
    boolean contains(String name);
    boolean contains(DataElement element);
    DataElement get(String name) throws Exception;
    Iterator iterator();
    List getNameList();
    DataElement put(DataElement obj) throws Exception;
    DataElement remove(String name) throws Exception;

    void addDataCollectionListener(DataCollectionListener l);
    void removeDataCollectionListener(DataCollectionListener l);

    String getCompleteName();
    void close() throws Exception;
    void init();
}

```

Properties

size – the number of DataElements in this DataCollection

dataElementType – Java meta class with a type of DataElements in this Data Collection

mutable – Boolean property indicating that Data Collection is mutable i.e. elements in it can be added, modified or removed

info – meta information about this Data Collection (see DataCollectionInfo below)

completeName – a unique name of Data Collection which defines position of Data Collection in Repository for the hierarchically organized set of Data Collections

nameList – non modifiable list of the names of all DataElements in this Data Collection

Methods

init – this method is invoked only once when Data Collection was just created and should perform all necessary actions for initialization of this Data Collection

close – this method releases all the resources used by the Data Collection. After invocation of ‘close’ none of Data Collection’s methods should be accessed.

contains – this method is to check whether the DataElement belongs to a given DataCollection. There are two forms of this method – one is used with instances of DataElement, another requires a name of DataElement to be passed

put – add or replaces DataElement to/in the DataCollection

remove – removes DataElement identified by its name for a Data Collection

get – retrieves references to an instance of DataElement via its name

iterator – returns an iterator which allows to cycle through all the DataElements in the DataCollection

addDataCollectionListener – registers with this DataCollection a listener which will receive notifications on various events happening inside DataCollection such as adding, removing and modifying of DataElements.

removeDataCollectionListener – de-registers listener of events in this Data Collection.

4.4 DataCollectionInfo

In many situations when dealing with various Data Collections, application developers may not have information about of the nature of information in a Data Collection. In the other situations it can be necessary to deal with multiple Data Collections and even if the nature of data in the Data Collection is known, using of such information on ad-hoc basis may sufficiently complicate application logic.

In order to solve this problem the concept of metadata (DataCollectionInfo) for a DataCollection is introduced. This metadata allows application developers to deal with Data Collection without knowing anything about its data. This is especially useful when dealing with hierarchically organized data (see Repository below). Such data can be visualized and processed on a regular basis.

DataCollectionInfo is defined as follows:

```
public final class DataCollectionInfo
{
    public Properties getProperties();
    public String getDisplayName();
    public boolean isVisible();
    public boolean isVisibleChildren();
}
```

```

    public ImageIcon getNodeImage();
    public ImageIcon getChildrenNodeImage();
    public Comparator getComparator();
    public QuerySystem getQuerySystem();
    ...
}

```

The properties of `DataCollectionInfo`:

properties – Data Collection are often created by `CollectionFactory` using information from a special configuration file. This file defines useful information in form of name-value pairs which information is stored in this property

displayName – human readable name with which `DataCollection` must be presented in Repository Pane

visible – whether Data Collection should be visible in Repository Pane

visibleChildren – whether Data Elements in this Data Collection should be visible in Repository Pane

nodeImage – icon that should be used to show `DataCollection`'s node (as expandable Data Element in another `DataCollection`)

childrenNodeImage – icon that should be used to show `DataElement`'s nodes of this `DataCollection` (as expandable Data Element in another `DataCollection`)

comparator – comparator that is used to sort `DataElements` of this `DataCollection` when showing them on repository tree

querySystem –

`DataCollectionInfo` provides also a number of utility methods to facilitate various housekeeping tasks for Data Collections.

```

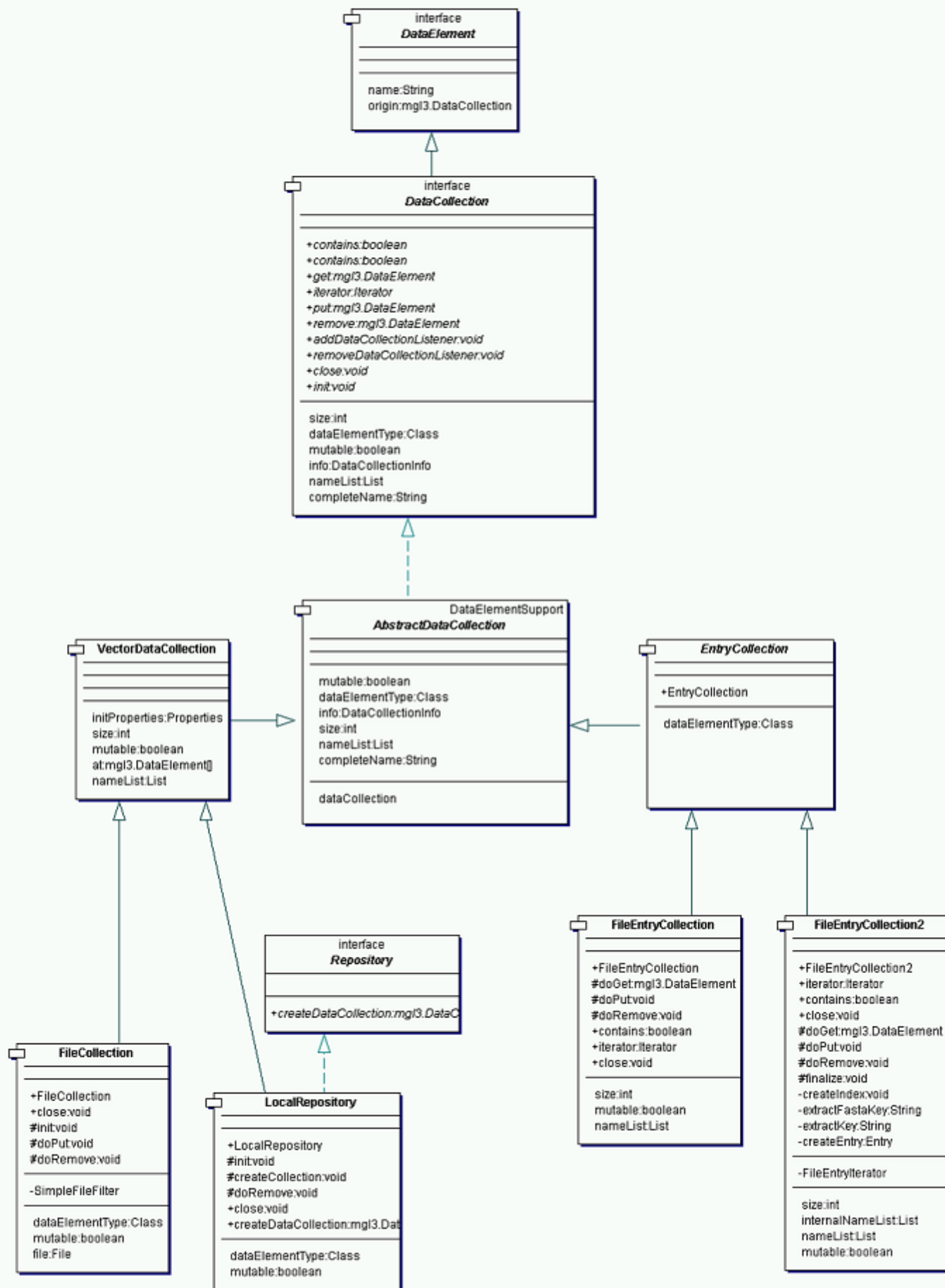
public List getUsedFiles();
public void addUsedFile(File file);

```

As it was described, Data Collection is usually defined by configuration files. Very often to speed processing Data Collection's data may be indexed and indexes stored in some separate files. This means that Data Collection can be comprised from several physical files on disk. When Data Collection is removed, it is necessary to remove these files also and two methods listed above are intended to keep track of files related to this Data Collection.

4.5 Usage of Data Collections

There are following important usages of Data Collection methodology. The diagram below shows the classes which bear Data Collection semantics.



4.6 Mutable and immutable Data Collections

Event processing

Event processing in DataCollections is similar to the concept of triggers in the most of traditional relational DBMS'. The mechanism allows application developers

- Receive notifications when data in Data Collection are about to change;
- Validate changes before they occur and if necessary reject them;
- Perform post processing after actual changes were made

Application component that is required to process events in Data Collections must implement interface DataCollectionListener which is defined as follows:

```
public interface DataCollectionListener extends EventListener
{
    void elementAdded(DataCollectionEvent e) throws Exception;
    void elementWillAdd(DataCollectionEvent e) throws DataCollectionVetoEx
    void elementChanged(DataCollectionEvent e) throws Exception;
    void elementWillChange(DataCollectionEvent e) throws DataCollectionVet
    void elementRemoved(DataCollectionEvent e) throws Exception;
    void elementWillRemove(DataCollectionEvent e) throws DataCollectionVet
}
```

Methods of this interface have the following meaning:

elementWillAdd – invoked before new DataElement will be added into Data Collection. If application decides to reject the changes, it must throw DataCollectionVetoException.

elementAdded – invoked after DataElement was added into Data Collection. Upon receiving this event application may, for example, update UI elements with new information

elementWillChange – invoked before modifying DataElement in a Data Collection. If application decides to reject the changes, it must throw DataCollectionVetoException.

elementChanged – invoked after DataElement in a Data Collection was changed.

elementWillRemove – invoked when a DataElement is about to be removed from a Data Collection. If application decides to reject the changes, it must throw DataCollectionVetoException.

elementRemoved – invoked after DataElement was removed from a Data Collection.

When processing events from DataCollection an application receives events that are instances of DataCollectionEvent which is defined as follows:

```
public class DataCollectionEvent extends EventObject
{
```

```

public static final int ELEMENT_ADDED      = 0;
public static final int ELEMENT_CHANGED   = 1;
public static final int ELEMENT_REMOVED   = 2;

private int type;
private DataElement dataElement;

public int getType();
public DataElement getDataElement();
public DataCollectionEvent(Object source, int type, DataElement dataEl
}

```

DataCollectionEvent defines the following properties:

type—a type of changes in data collection—i.e. whether element is added, modified or removed

dataElement—reference to an instance DataElement that is added, modified or removed

Cache

Perhaps the most often used operation on Data Collections is locating a DataElement via its name using **contains (String name)** or **get** methods. In order to speed up execution of these primitives, default implementation of DataCollection interface is provided which maintains internal hash table of mapping between names of DataElements and actual instances.

It is also important that when a DataElement is looked up in DataCollection, the **get** method would return the same instance of DataElement instead of creating new Data Element every time. Internal hash table stores such instances and re-uses them when necessary.

When an application tries to locate a DataElement in Data Collection, the default implementation looks first in the internal hash table and if element is not found fetches data information from the physical location. Otherwise instance of Data Element stored in the cache is returned.

This default behavior is implemented in AbstractDataCollection class as follows:

```

abstract public class AbstractDataCollection extends DataElementSupport implements
{

    public DataElement get(String name) throws Exception
    {
        DataElement de = (DataElement)v_cache.get( name );
        if ( de==null )
        {
            de = doGet( name );
        }
    }
}

```

```

        if ( de != null )
            cachePut( de );
    }
    return de;
}

protected DataElement doGet(String name) throws Exception;
...
}

```

All Data Collections that required to have cached storage are recommended extend `AbstractDataCollection` class and redefine `doGet` method which will be invoked when fetching of `DataElement`'s data from a physical storage is required.

Values in the cache are stored via soft references. This means that in response to lack of memory JVM will clear the cache automatically without additional efforts for an application developer.

`AbstractDataCollection` implements also `DataElement` interface. This way Data Collections that are based on `AbstractDataCollection` can be used as `DataElements` in other Data Collections thus providing hierarchical organization of the data.

Logging

Data Collections have internal logging mechanism which is based LOG4J framework from Jakarta project. Default implementation `AbstractDataCollection` assigns a logging category which can be used by particular implementations to reports various information during data processing.

The category is assigned as follows:

```

protected void initLog()
{
    String name = "access." + getCompleteName();
    cat = Category.getInstance(name);
}

```

Therefore, in order to intercept logging info from a particular `DataCollection` instance, for example, `localhost.profiles`, a user must define somewhere in the log configuration file the following entries:

```

log4j.category.access.localhost.profiles    =ERROR,A1
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.File=./log/access_localhost_profiles
...

```

This will define logging preferences for all messages with level ERROR or higher (such messages should be redirected to the file ./log/access_localhost_profiles).

4.7 CollectionFactory

CollectionFactory is a factory class which provides methods for managing Data Collections. It allows application developer to:

- Create new Data Collection
- Get DataCollection's instance via its complete name
- Create a copy of existing Data Collection

In order to create new Data Collection's instance from its data stored on disk the following method is provided:

```
static public DataCollection createCollection(DataCollection parent, Properties properties) throws Exception;
```

parent - is an instance of parent DataCollection i.e. where newly created Data Collection will be put to

properties – is a set of name-value pairs that contains necessary information for creating DataCollection. It is the same set of properties that is stored in **properties** property of the DataCollectionInfo for this Data Collection (see above).

The following values can be passed as properties when creating new Data Collection (they are defined in DataCollection interface). Depending on the type of Data Collection some properties may be optional but 'name' and 'class' are required for all Data Collections.

Name	Description
name	Required. Internal name of the Data Collection. This is obviously the name of it as of a Data Element – therefore it must be unique for the same parent
class	Required. Class name that must be instantiated in order to produce a new instance of DataCollection
configPath	Property where name of Data Collection's config file's path is stored
configFile	Property where name of Data Collection's config file is stored
filePath	Property where path name of Data Collection's main file is stored. For example, during import it is not required that Data Collection's file will be physically copied into repository. In the case this property will keep path to the directory where the file is located
file	Name of Data Collection's main file

node-image	File name for the image that represents DataCollection's node in Repository Pane
childrenNodeImage	File name for the image that represents DataCollection's DataElements in Repository Pane
nodeVisible	This Boolean property is used for defining whether current node will be displayed in a Repository Pane
childrenNodeVisible	This Boolean property is used for defining whether DataElement leafs of current node will be displayed in a Repository Pane
transformer	Property for storing a class of the transformer which is used to transform entries into Java objects
job-control	Object that is used for reporting a progress on various time-consuming operations in Data Collections
nextConfig	Property for storing name of primary data collection config file. It is used for TransformedDataCollections to specify primary data collection
comparator-object	Comparator which is used to sort elements of Data Collection

Most of the values listed above are read from configuration files stored on disk.

4.8 Basic Data Collections

Repository

Repository is a special form of Data Collection that creates hierarchy of Data Collections. In BioUML a special form of Repository called LocalRepository is used. LocalRepository creates hierarchy of Data Collection mimicking hierarchy of local filesystem where each subdirectory is Data Collection that can contains other Data Collections.

Local Repository is defined as follows:

```
public class LocalRepository extends VectorDataCollection implements Repos
{
    protected File root;
    public DataCollection createDataCollection(String name, Properties prop
    protected void createCollection(File file, boolean isNotify, FunctionJob
    ...
}
```

Method **createDataCollection** is used to create new Data Collection on disk. If necessary it copies files into repository. This method is used in BioUML when importing sequences, matrices and so on.

Since LocalRepository operates on a top of directory structure it needs a mechanism to extract DataCollectionInfo for a given directory and use it to interpret a files in directory in order to create Data Elements for the DataCollection. For this the following principle is used.

When starting processing a directory LocalRepository looks for default.config file. This file contains name of the Data Collection, its class and icons. For example the following configuration file is used for 'sequences' node of Repository.

```
class = ru.biosoft.access.LocalRepository
name = sequences
entry.delimiters = " "
node-image = sequences.gif
childrenNodeImage = sequenceSet.gif
displayName = Sequences
```

Then LocalRepository iterates through all *.node.config files and interprets them as Data Elements for the given Data Collection ('sequences' in our example). Suppose this search returns the following files:

```
embl.node.config
fasta.node.config
gbembl.node.config
genbank.node.config
```

Therefore the 'sequences' Data Collection will contain 4 elements defined by content of each .node.config. Since each node is also a Data Collection, the .node.config files will be used to extract DataCollectionInfo and create hierarchy of DataCollection further.

Examples of .node.config files are provided below.

AbstractDataCollection

The purpose and implementation details of AbstractDataCollection were already described. Briefly, AbstractDataCollection handles common tasks necessary for most Data Collections and provide caching mechanism to speed up looking up of the Data Collection's elements as well reusing of the DataElements once they were created.

FileEntryCollection

FileEntryCollection and FileEntryCollection2 are primary Data Collections for dealing with biological databases which are flat files containing multiple plain-text entries (see Entry above).

The classes redefine **doGet** method (method which performs physical fetching of the entry's data from a flat file).

Both types of Data Collections index the entries in flat files for faster access. The difference between them is in the type of index that they are using:

FileEntryCollection uses hash indexes that are created in memory for looking up the offsets of Entries in the flat file

FileEntryCollection2 uses B-tree indexes on disk to look up offsets of the Entries on a flat file.

VectorDataCollection

This is a type of Data Collection that uses java.util.Vector as its storage mechanism for the Data Element.

VectorDataCollections are extensively used as in-memory Data Collections which have a limited number of elements that are often updated.

VectorDataCollection redefines **put** method in order to keep DataElements always ordered using their names in the vector. It also redefines **doGet** method to perform binary search on the list of DataElements (since it is kept sorted at all times).

FileCollection

FileCollection is an extension of VectorDataCollection for processing of the list of files. It can be used to represent file system and query it using Data Collection interface.

Additional property 'fileFilter' can be passed with **properties** when creating new FileCollection. In this case, this property will be used to filter files according to their extensions.

For instance if 'fileFilter=*.gif;*.jpg', FileCollection can be used to browse through all the GIF and JPEG images present in the file system.

DataCollectionUnion

DataCollectionUnion can be thought of as an analogue of UNION clause in SQL. It allows application developers to process several DataCollections simultaneously as if it would be the one.

When using **get** method to fetch a DataElement via its name, it is a responsibility of an application to ensure that all DataElements from all included DataCollections have globally unique names. DataCollectionUnion doesn't attempt to check or maintain this uniqueness – it simply returns the first DataElement with a given name which is present in any of the associated DataCollections.

The list of DataCollections to be united must be itself passed as a DataCollection to the constructor of DataCollectionUnion (i.e. it will be an origin).

Transformed Data Collections

It was specified that one of the primary reasons of Data Collections framework is to provide a mechanism to map plain text data from biological databases into Java objects for more efficient processing. This functionality is provided by Transformed Data Collections.

Derived Data Collection

Transformed Data Collections are based on a concept of Derived Data Collection. DerivedDataCollection itself is merely a “wrapper” delegating all operation on it to the actual Data Collection it wraps. The reason for introducing DerivedDataCollection is to provide different Data Collection object which can be modified without affecting the primary Data Collection.

Conceptually Derived Data Collection follows *Decorator* design pattern.

Transformer

Transformed Data Collections must be initialized using ‘transformer’ property (see above). This property specifies name of Java class that transforms entries from biological databases from plain text format to Java objects and vice versa.

Transformer interface is defined as follows:

```
public interface Transformer
{
    Class getInputType();
    Class getOutputType();
    DataElement transformInput(DataElement input) throws Exception;
    DataElement transformOutput(DataElement output) throws Exception;
    ...
}
```

Where properties:

inputType— Java class specifying type of input Data Elements i.e. those to be transformed. For most transformers it is `ru.biosoft.access.Entry.class`

outputType— Java class specifying type of output Data Elements i.e. those to be produced.

Methods:

transformInput— methods that transform DataElements of the input type to the Data Elements of the outputType

transformOutput— methods that transform DataElements of the output type to the Data Elements of the inputType

Therefore Transformed Data Collection can be defined as:

- It extends Derived Data Collection
- It adds transformers for object-to-plaintext mapping

The most important functionality Transformed Data Collection provides is a redefinition of **doGet** and **doPut** to exchange data with a primary Data Collection which contains raw plain-text entries.

```
public DataElement doGet( String name ) throws Exception
{
    DataElement de = super.doGet( name );
```

```

        if( de!=null )
            de = transformer.transformInput( de );
        return de;
    }

    public void doPut( DataElement element ) throws Exception
    {
        puttedElement = element;
        DataElement tde = transformer.transformOutput( element );
        collection.put( tde );
        puttedElement = null;
    }

```

4.9 Filtering Data Collections

Very often Data Collections contain too many entries and application requires only small subset for example for visualization. In order to deal with such situation another type of Derived Data Collections is introduced – FilteredDataCollection.

Filtered Data Collections must be initialized using ‘filter’ property. This property specifies an instance of Java class implementing Filter interface that specifies whether DataElement should be screened out or not.

Filter interface is defined as follows:

```

public interface Filter
{
    Filter INCLUDE_ALL_FILTER = new IncludeAllFilter();
    Filter INCLUDE_NONE_FILTER = new IncludeNoneFilter();
    boolean isEnabled();
    boolean isAcceptable(DataElement de);
    ...
}

```

Actual filtering functionality is implemented in **isAcceptable** method which defines which Data Elements should be included.

Filters also have **enabled** property to indicate whether a filter specified for a Filtered Data Collection should be used at all.

When Filtered Data Collection is initialized it uses filter to generate the list of names of all Data Elements that satisfied filtering criteria. Then method **get** is redefined to use only DataElements from this list as follows:

```

public boolean contains(String name)
{
    if( sorted )

```

```

        {
            return (Collections.binarySearch(filteredNames,name) >= 0);
        }
        return filteredNames.contains(name);
    }
    public DataElement get(String name) throws Exception
    {
        if(!contains(name))
            return null;
        return super.get(name);
    }
}

```

Special filters

MutableFilter. This filter is introduced to facilitate using of filters in the UI controls. The filter extending `MutableFilter` is supposed to be dynamically enabled or disabled. When it happens, the filter fires `propertyChange` event thus allowing any entity to re-apply filtered information (generally event listener creates new (recreates) `FilteredDataCollection` for this purpose).

CompositeFilter. In many situations there are several filters to be applied to a `Data Collection`. In such cases `CompositeFilter` can be used – which is constructed with an array of filters to be applied. `DataElement` is accepted if and only if it is accepted by all the associated filters of the `CompositeFilter`.

PatternFilter. This is an abstract class to facilitate implementation of filters that use Perl5 regular expressions applied to the properties of `DataElements` for selection. Many composite filters use collection of `PatternFilters` to select `DataElements` using various types of their properties. Concrete `PatternFilters` must define the following method in order to return value of the property to which filter must be applied.

```

abstract public String getCheckedProperty(DataElement de);

```

For instance, `SiteFilter` provides the following `PatternFilter`

```

public static class TypeFilter extends PatternFilter
{
    public String getCheckedProperty(DataElement de)
    {
        if(de instanceof Site)
            return ((Site)de).getType();
        return null;
    }
}

```

4.10 QuerySystem



Using of indexes can greatly improve performance for some DataCollection operations. To formalize using of indexes with data collections we introduce the concept of QuerySystem that provides access to all indexes associated with particular DataCollection.

QuerySystem interface is defined as follows:

```

public interface QuerySystem extends DataCollectionListener
{
    public static final String QUERY_SYSTEM_CLASS = "querySystem";
    public static final String INDEX_BLOCK_SIZE = "indexBlockSize";

    public Index[] getIndexes();
    public Index getIndex(String name);
    public void close();
}

```

Methods:

getIndexes - return all indexes associated with this DataCollection and supported by this QuerySystem.

getIndex – returns index by its name. Generally QuerySystem subclasses defines string constants to get particular Index.

close – close all indexes to release used resources and files.

QuerySystem is quite independent and specific entity.

Independent means:

- QuerySystem subclasses can be associated with any DataCollection. Corresponding QuerySystem instance will be created by DataCollectionInfo using relevant information from config file. Then other methods can get and use this QuerySystem calling **dataCollection.getInfo().getQuerySystem()** method.
- DataCollection should not bother about QuerySystem indexes updating when data collection content is changing. For this purpose QuerySystem registers itself as DataCollectionListener and its responsibility to update indexes.

Specific means:

- indexes, their type and semantics depends from particular QuerySystem subclasses.
- generally QuerySystem subclass corresponds to some DataElement subclass and this QuerySystem indexes corresponds to some DataElement subclass properties. For example SiteQuerySystem is used for data collection that stores sites as data elements and SiteQuerySystem provides indexes ‘from’ and ‘to’ that corresponds to from and to property of site.
- QuerySystem can be used by different entities to solve different particular tasks. For example, SequenceTransformer uses SequenceQuerySystem to optimize sequences loading from data collection but SiteQuerySystem is used by RegionFilter to optimize query processing (selection of site set for the given map region).

Creating QuerySystem

QuerySystem interface defines two string constants for properties that can be specified in config file for corresponding DataCollection to initialize QuerySystem subclass:

QUERY_SYSTEM_CLASS – specifies QuerySystem subclass

INDEX_BLOCK_SIZE – block size for indexes used by QuerySystem.

Using this information DataCollectionInfo creates corresponding QuerySystem subclass and registers all index files (it is essential to correctly remove index files when corresponding data collection is removing from repository).

```
// code in DataCollectionInfo to initialize QuerySystem
String className = properties.getProperty(QuerySystem.QUERY_SYSTEM_CLASS);
if( className != null )
{
    Class c = Class.forName(className);
    Class[] params = {DataCollection.class};
    java.lang.reflect.Constructor constructor = c.getConstructor(params);
    Object[] args = {dc};
    querySystem = (QuerySystem) constructor.newInstance(args);

    // register index files
    Index[] indexes = querySystem.getIndexes();
}
```



```

        if( indexes != null )
        {
            for(int i=0; i<indexes.length; i++)
                addUsedFile( indexes[i].getIndexFile() );
        }
    }
}

```

Important!

- 1) To be correctly initialized by DataCollectionInfo all QuerySystem subclasses should have public constructor with DataCollection argument. For example,
- 2) For indexes updating when corresponding DataCollection content is changing, QuerySystem should register itself as DataCollectionListener.

This is an example of correct QuerySystem subclass constructor:

```

public class SiteQuerySystem implements QuerySystem
{
    public SiteQuerySystem(DataCollection dc)
    {
        ...
        dc.addDataCollectionListener(this);
        ...
    }
    ...
}

```

Closing QuerySystem

QuerySystem is automatically closed when DataCollection is closed. Corresponding code is provided by AbstractDataCollection **close** method.

```

public void close() throws Exception
{
    QuerySystem querySystem = info.getQuerySystem();
    if( querySystem != null )
        querySystem.close();
    ...
}

```

Generally when QuerySystem is closed it is invalid.

Updating indexes

QuerySystem is responsible for indexes updating when DataCollection content is changing. For this purpose QuerySystem will receive the corresponding DataCollection in its constructor and should register itself as DataCollectionListener. Listening DataCollectionEvents QuerySystem may (should) update the indexes.

Different QuerySystem subclasses can have different policies for indexes updating. Some of them may suggest (or know) that DataCollection content is unchangeable and ignore DataCollectionEvents.

Indexes

There are three main classes supporting indexes concept.

Index – interface for storing/extracting indexes. This interface extends **java.util.Map** interface.

BTreeIndex – implementation of Index interface. BTrees are used for optimal access to indexes using file system.

Note: index cannot contain duplicate keys.

BTreeRangeIndex – implementation of BTreeIndex where keys can be duplicated.

Note: For more details see javadoc for this classes.

5 Plug-in development

Enter topic text here.

5.1 Create new plug-in

Step1. Create plugin description.

Add new folder to “plugins” (for example, bdk\plugins\biouml.examples.sql_0.8.6). Create plugin.properties and plugin.xml files.

plugin.properties is a simple properties file. In our example bdk\plugins\biouml.examples.sql_0.8.6\plugin.properties:

```
pluginName = MyDatabase
providerName = <your_company_name>
```

plugin.xml is a plugin description in XML format. In our example bdk\plugins\biouml.examples.sql_0.8.6\plugin.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="%pluginName"
  id="biouml.examples.sql"
  version="0.8.6"
  provider-name="%providerName">
  <requires>
    <import plugin="org.apache.log4j"/>
    <import plugin="biouml.workbench"/>
    <import plugin="ru.biosoft.access"/>
    <import plugin="ru.biosoft.workbench"/>
    <import plugin="com.beanexplorer"/>
  </requires>
  <runtime>
    <library name="sql_example.jar">
      <export name="*" />
    </library>
  </runtime>
</plugin>
```

Step2. Create new Ant target for new plugin.

Open bdk\src\build.xml file. Add new <target> to <project>. For example,

```
<target name="plugin.examples.sql" depends="local_settings, compile">
```

```

<echo message=" MyDatabase plugin" />
<jar jarfile="${PLUGIN_DIR}/biouml.examples.sql_${VERSION}/sql_example.jar">
  <fileset dir="${OUTDIR}">
    <patternset>
      <include name="biouml/examples/sql/**/*.class" />
      <exclude name="biouml/examples/sql/_test/**/*.class" />
    </patternset>
  </fileset>
  <fileset dir="${SRCDIR}">
    <patternset>
      <include name="biouml/examples/sql/**/*.gif" />
      <include name="biouml/examples/sql/**/*.html" />
      <include name="biouml/examples/sql/**/*.png" />
    </patternset>
  </fileset>
</jar>
</target>

```

This target compile all plugin classes exclude tests and pack it into sql_example.jar file in plugin folder, pictures and .html files are also added to this JAR. The name of the .jar file must be the same as library name in plugin.xml

Step3. Compiling.

Go to bdk\src folder and run created target (in our example, "plugin.examples.sql"). You should see BUILD SUCCESSFUL at the end of process. If you'll see BUILD FAILED check all steps, correct errors and try again.

5.2 How to create SQL transformer

Create new package for your plugin. In examples, plugin packages are subpackages of biouml.examples package (look at bdk\examples directory). Put all your java classes in this package.

Create new transformer class for SQL table. For SqlDataCollection transformer should implements ru.biosoft.access.SqlTransformer interface or extends ru.biosoft.access.SqlTransformerSupport class. Look at sources or JavaDoc for more information about SqlTransformer. In our example we use next SQL table for molecules

```

CREATE TABLE `molecules` (
  `ID` varchar(20) NOT NULL default '',
  `title` varchar(100) NOT NULL default '',
  `completeName` varchar(200) default NULL,
  `description` text,
  `comment` text,
  UNIQUE KEY `IDX_UNIQUE_concepts_ID` (`ID`)
)

```

Create biouml.examples.sql.MyMoleculeSqlTransformer

```
package biouml.examples.sql;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import biouml.standard.type.Substance;
import ru.biosoft.access.DataCollection;
import ru.biosoft.access.DataElement;
import ru.biosoft.access.SqlTransformerSupport;

public class MyMoleculeSqlTransformer extends SqlTransformerSupport
{
    /* Transformer init method */
    public boolean init(DataCollection owner)
    {
        table = "molecules"; //TODO: put your SQL table name here
        this.owner = owner;
        return true;
    }

    /* Return collection element class*/
    public Class getTemplateClass()
    {
        return Substance.class;
    }

    /* Return select query to get information for one element*/
    public String getSelectQuery()
    {
        return "SELECT id, title, description, comment, completeName " + " ";
    }

    /* Creates data element by ResultSet*/
    public DataElement create(ResultSet resultSet, Connection connection)
    {
        //TODO: change this method to fill data element fields by result set
        Substance molecule = new Substance(owner, resultSet.getString(1));
        molecule.setTitle(resultSet.getString(2));
        molecule.setDescription(resultSet.getString(3));
    }
}
```

```

        molecule.setComment(resultSet.getString(4));
        molecule.setCompleteName(resultSet.getString(5));

    return molecule;
}

/* Creates INSERT query for SQL table by DataElement*/
public void addInsertCommands(Statement statement, DataElement de) throws
{
    //TODO: change this method for your SQL table structure
    StringBuffer result = new StringBuffer("INSERT INTO " + table + "

    Substance molecule = (Substance)de;
    result.append(validateValue(molecule.getName()));
    result.append(", " + validateValue(molecule.getTitle()));
    result.append(", " + validateValue(molecule.getDescription()));
    result.append(", " + validateValue(molecule.getComment()));
    result.append(", " + validateValue(molecule.getComment()));

    result.append(")");
    statement.addBatch(result.toString());
}

/* Creates DELETE query for SQL table to delete DataElement by name*/
public void addDeleteCommands(Statement statement, String name) throws
{
    statement.addBatch("DELETE FROM " + table + " WHERE id='" + name + "

}

(get this example at bdk\examples\databases\sql\src\biouml\examples\sql\MyMoleculeSqlTransformer.
java)

```

Read [Create new plug-in](#) section to configure new plugin for BioUML.

6 JavaScript

Enter topic text here.

6.1 Functions and host objects

JavaScript plug-in defines two extension points that allows other plug-ins to expose their functions and objects to Shell mode , JavaScript tab or JavaScript documents.

ru.biosoft.plugins.javascript.function

This extension point allows plug-in to contribute its JavaScript functions. This JavaScript functions will be shown in **Analyses/JavaScript/Functions** section in repository tree. Function help will be shown in View/Edit tab when the function item will be selected in repository tree.

ru.biosoft.plugins.javascript.hostObject

Using this extension point plug-in can provide access to particular Java objects (host objects) provided by plug-in. Plug-in host objects will be shown in **Analyses/JavaScript/Host objects** section in repository tree. Host object description (help) will be shown in View/Edit tab when the host object item will be selected in repository tree.

Standard host objects

- R - facade for R usage.
- data - facade for data-manipulations.
- dataFilter - facade for data-filtering.
- microarray - facade for microarray analysis.
- sbw - host object for SBW integration.

Documentation

Using described above extension points developer can provide description (documentation) for his functions and host objects.

All functions and host objects for which help description is available will be shown in plug-in tree as children of JavaScript plug-in. When corresponding function or host object will be selected, its description will shown in Property Inspector pane.

Alternatively user can type `function_name` or `help(function_name)` and function description will be printed in JavaScript shell.

6.2 R JavaScript host object

R is an integrated suite of software facilities for data manipulation, calculation and graphical display (<http://www.r-project.org>).

BioUML supports R script by **R JavaScript** host object.

R methods

- **local** - return RObject for locally installed R application using Java R Interface (JRI API). It's the fastest way for using R.

Note: `R_HOME` environment variable should be set.

Example:

```
var rObject = R.local();
```

- **rserve** - return RObject for locally installed R application using RServe service. This type of RObject is more stable and recommended to use with locally installed R application

Note: If RServe is not running the system will try to run it automatically, otherwise you should run it manually.

Example:

```
var rObject = R.rserve();
```

- **connect** - return RObject for remote R installation. In this case R should be installed on BioUML server. This method have two string parameters:

host - address of BioUML server,

port - connection port

Example:

```
var rObject = R.connect("server.biouml.org", 80);
```

Note: read User Guide for more information about R support in BioUML.

7 Examples

This topic describes BDK examples

7.1 SQL database

Sources: bdk\examples\databases\sql

This example demonstrates how integrate SQL database into BioUML workbench/server.

Structure of the directory

- data - database configuration folder. Look [SQL database configuration](#) for details.
- plugins - complete set of plug-in description files. Read [Create new plug-in](#) to do the same things manually.
- src - source Java files for plugin and building scripts.
 - build.properties - properties for build process (put your SQL database preferences here)
 - build.xml - ant target description
 - data.sql - test database tables INSERT commands
 - tables.sql - test database tables CREATE commands

How to run

Go to **bdk\examples\databases\sql\src** folder. build.xml file defines a set of targets. Main targets are:

- create.db - create necessary tables in database
- plugin.sql - compile and deploy plugin
- run - run biouml application
- test.molecule - run test for "molecule" collection
- test.relation - run test for "relation" collection

7.1.1 Overview

Generally integration of SQL database into BioUML framework consists of 4 steps:

1. **Basic plug-in** - provides basic functionality: show the database content and full text search. The database will be available only locally.
 - 1.1. create Java classes (SQL transformers) that maps content of the database into corresponding Java objects. BioUML contains a set of predefined Java classes (package biouml.standard.type) for main biological concepts like Molecule, Gene, Reaction, Protein.
 - 1.2. create plugin.xml and jar files with Java files from 1.1
 - 1.3. create config files that allows BioUML to connect to the database.
 - 1.4. specify fields to be indexed by Lucene (full text search engine)
2. **Diagrams** - during this step it is needed to describe how information from the database will be shown on diagrams.
 - 2.1. Diagram transformer - maps pathways/diagrams from the database into biouml.model.Diagram object (if the

database contains curated pathways)

- 2.2. Query engine - subclass of `biouml.model.QueryEngine` searches for related/linked components in the diagram. Using this class BioUML will allow to build and extend diagrams by user request.
3. **Graphic notation** - BioUML provides default graphic notation to display information about biological objects and their properties. However you can create your own graphic notation using BioUML graphic notation editor. http://www.biouml.org/demo/Graphic_Notations_Editor.exe
4. **BioUML server**
 - 4.1. Set up Tomcat and BioUML server
 - 4.2. Put configuration files for the database connections into the corresponding server directory
 - 4.3. Configure access rights (read/write)

See also:

http://www.biouml.org/demo/Graphic_Notations_Editor.exe

7.1.2 SQL database config file

All BioUML database descriptions stored in **data** folder. Go to **data** and create new folder for your database. For example, **data\mydatabase**. Top level description of each database stored in default.config file. Create default.config file for your database and open it in the text editor. Type name, class, module-type, next-config parameters, for example

```
name=MyDatabase
class=biouml.model.Module
module-type=biouml.standard.SqlModuleType
nextConfig=default.repository
plugins=biouml.examples.sql
```

(get this file at `bdk\examples\databases\sql\data\mydatabase\default.config`)

- name – name of database
- class – class for BioUML structure, use `biouml.model.Module` for top level of database
- module-type – module type class, use `biouml.standard.SqlModuleType` for SQL databases with simple structure
- nextConfig – name of the file with repository properties
- plugins – name of necessary plugin

Next step, create `default.repository` file and type there next parameters

```
name= MyDatabase
class=ru.biosoft.access.LocalRepository
```

(get this file at bdk\examples\databases\sql\data\mydatabase\default.repository)

- name—name of repository, should be the same as name in default.config file
- class—repository class

“Data” collection

Create “Data” (bdk\examples\databases\sql\data\mydatabase\Data) with default.config file (bdk\examples\databases\sql\data\mydatabase\Data\default.config). Put name and class parameters to default.config

```
name= Data
class= ru.biosoft.access.LocalRepository
```

This values define Data collection and all element of this collection should be defined with help of *.node.config file in Data folder. For example, create molecule.node.config file (bdk\examples\databases\sql\data\mydatabase\Data\molecule.node.config). This file should define “molecule” collection based on SQL table with concepts information.

```
name=molecule
data-element-class=biouml.standard.type.Molecule
class=ru.biosoft.access.SqlDataCollection
transformerClass= biouml.examples.sql.MyMoleculeSqlTransformer
jdbcDriverClass=org.gjt.mm.mysql.Driver
jdbcURL=jdbc:mysql://<host>:<port>/<database_name>
jdbcUser=<database_username>
jdbcPassword=<database_password>
```

(get this file at bdk\examples\data\databases\sql\data\mydatabase\Data\concept.node.config)

- name—collection name
- data-element-class—type of collection element. You can use standard BioUML types (look at biouml.standard.type package) or create your own type (type class must implements ru.biosoft.access.DataElement interface)
- class—collection class, use SqlDataCollection for SQL collections.
- transformerClass—SQL to DataElement transformer class.
- jdbcDriverClass—driver for JDBC connection
- jdbcURL—JDBC connection string, fill host, port and database_name in your configuration
- jdbcUser—database user
- jdbcPassword—database password

7.1.3 SQL transformers

This example consists of 2 SQL transformers: one for "molecule" collection (**bdk\examples\databases\sql\src\biouml\examples\sql\MyMoleculeSqlTransformer.java**) and another for "relation" collection (**bdk\examples\databases\sql\src\biouml\examples\sql\MyRelationSqlTransformer.java**)

You can create your own transformers. Read [How to create SQL transformer](#) for more information.

Testing

Standard way of classes and modules testing in BioUML is **JUnit** tests. Put your tests in **_test** package.

There are 2 tests in SQL database example:

- bdk\examples\databases\sql\src\biouml\examples\sql_test\MyMoleculeSqlTransformerTest.java
- bdk\examples\databases\sql\src\biouml\examples\sql_test\MyReelationSqlTransformerTest.java

To run this tests use special ant targets (**bdk\examples\databases\sql\src\build.xml**)

- test.molecule - run test for "molecule" collection
- test.relation - run test for "relation" collection