In this tutorial, we describe steps for setting up a Maven project that uses `libSBOLj` in Eclipse, and explain how we use SBOL 2.0 to represent the function of a state-of-the-art design, namely a CRISPR-based repression module Kiani *et al.* [1].

Set up Maven plugins in Eclipse

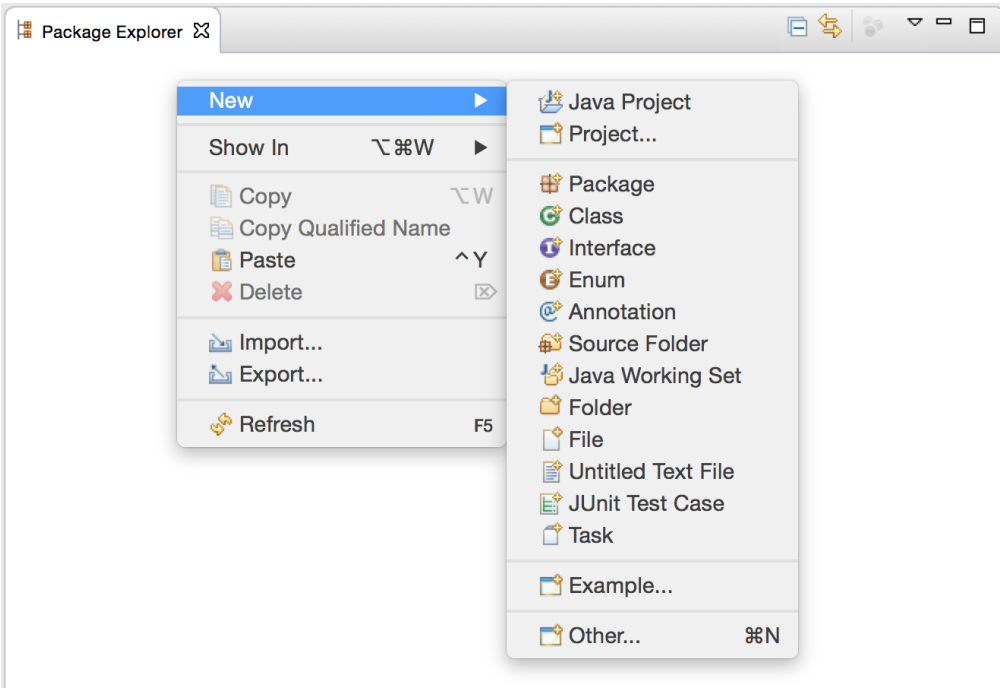
In this section, we describe steps for installing Maven plugins in Eclipse. The Eclipse version used is Luna Service Release 2 (4.4.2). Here are the steps:

1. In Eclipse, go to Help and select Install New Software,
2. Add a new software site: Name = slf4j, URL = `http://www.fuin.org/p2-repository/`,
3. Select this site to work with, expand Maven osgi-bundles, and select slf4j.api,
4. Click Next and follow the installation process,
5. Add a new software site: Name = Maven Plugins, URL = `http://download.eclipse.org/technology/m2e/releases`,
6. Select this site to work with, expand Maven Integration for Eclipse, and select m2e - Maven Integration for Eclipse, and
7. Click Next and follow the installation process.

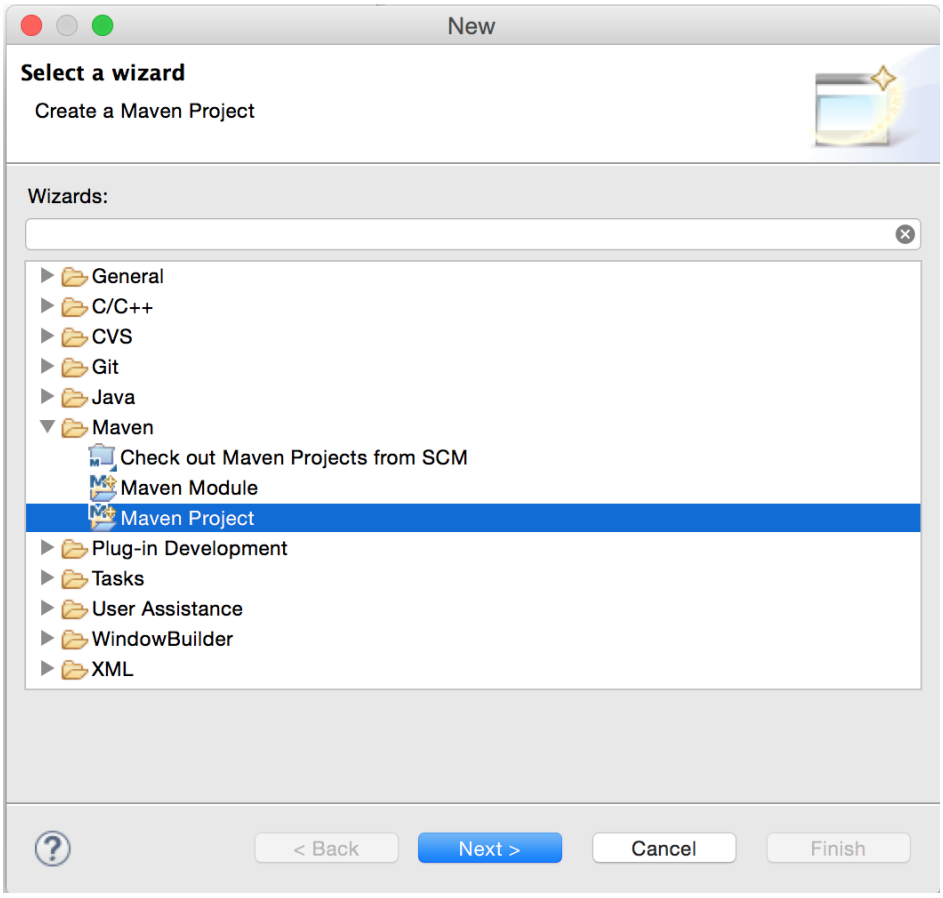
Creating a new project

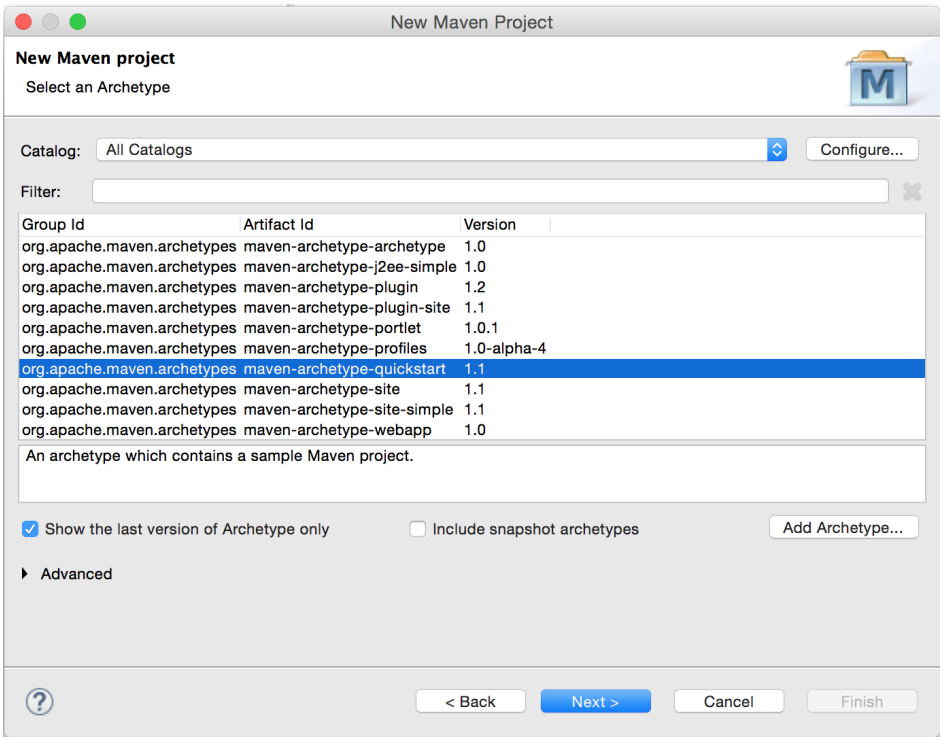
After the Maven plugin is installed, we are now ready to create a new Maven project in Eclipse. The following text describe the necessary steps.

In the package explorer window, right click and select **New** → **Other**.

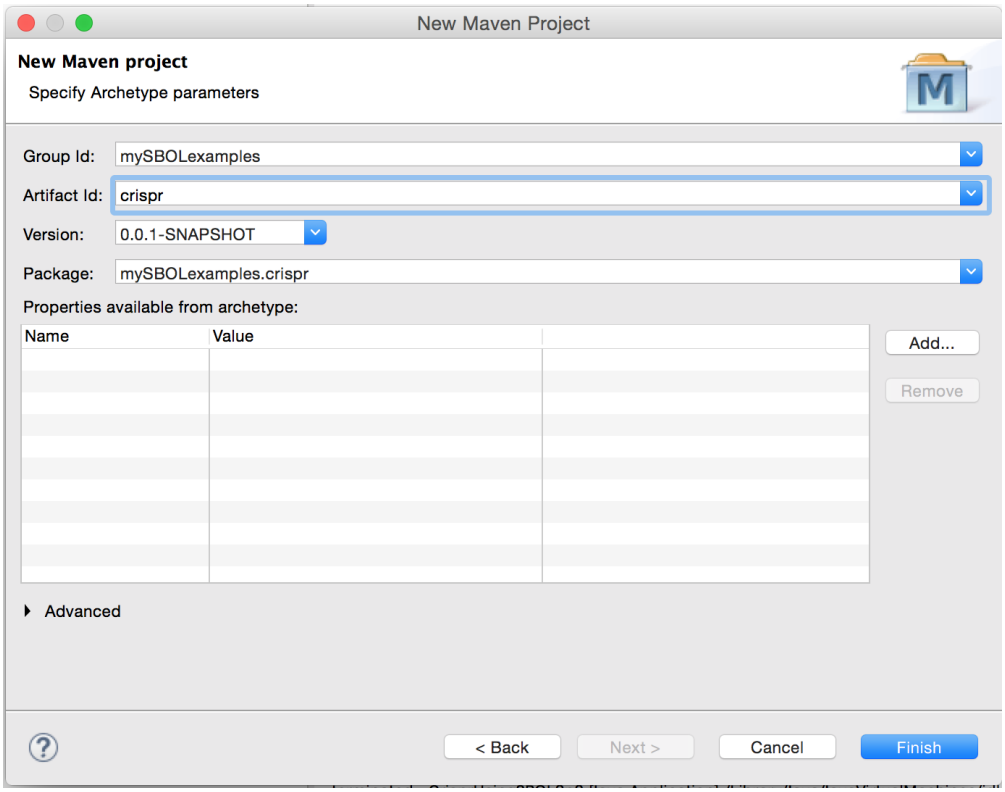


Under the **Maven** folder, choose **Maven Project**, and then follow the default options for the project setup.



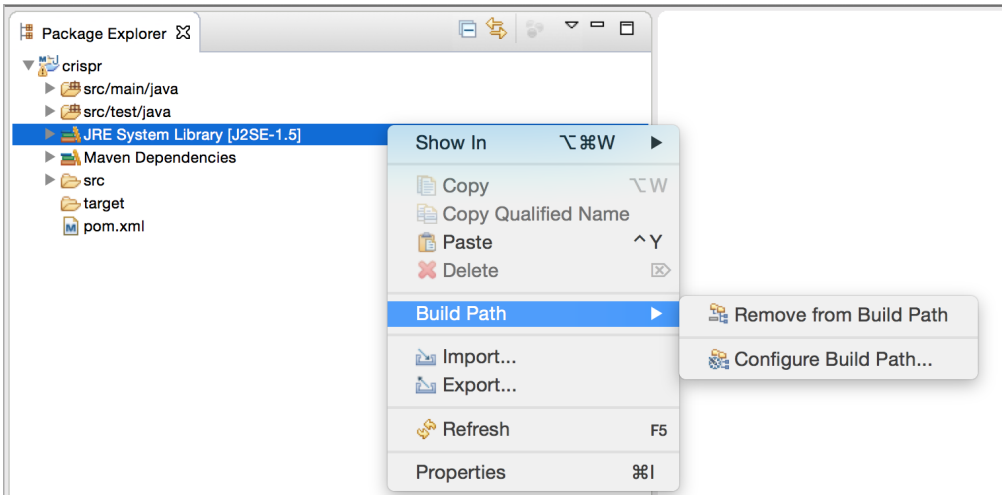


In the window for specifying archetype parameters, you may type your group ID and artifact ID. In this tutorial, they are specified as shown below.

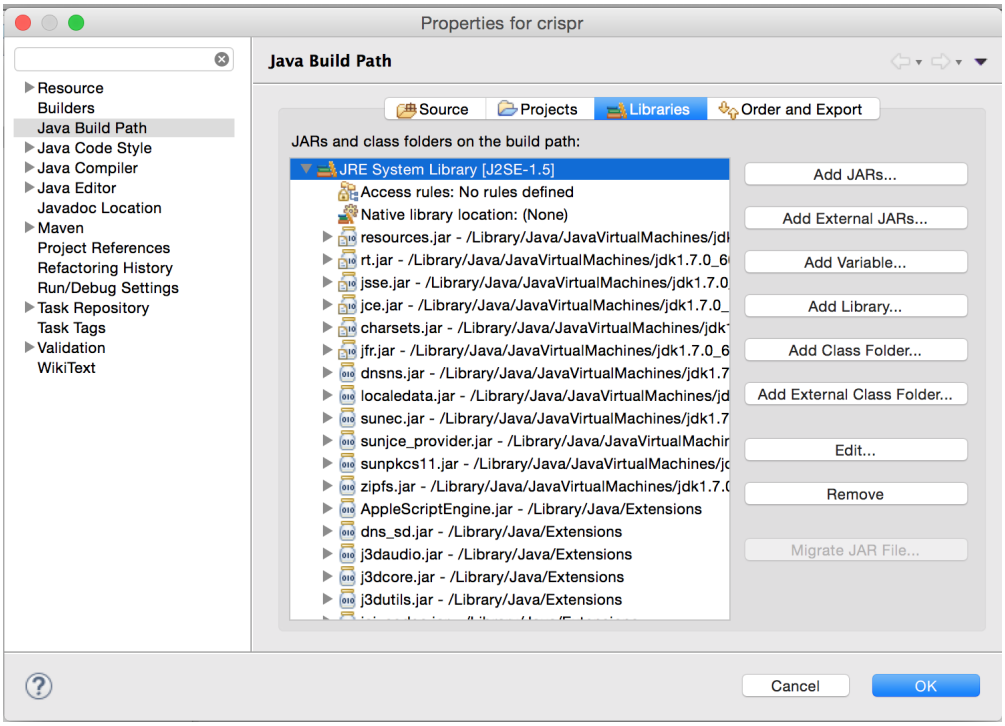


Once the project setup is finished, you should be able to see two Java source folders, a **JRE System Library** and a **Maven Dependencies** library, and a **pom.xml** file.

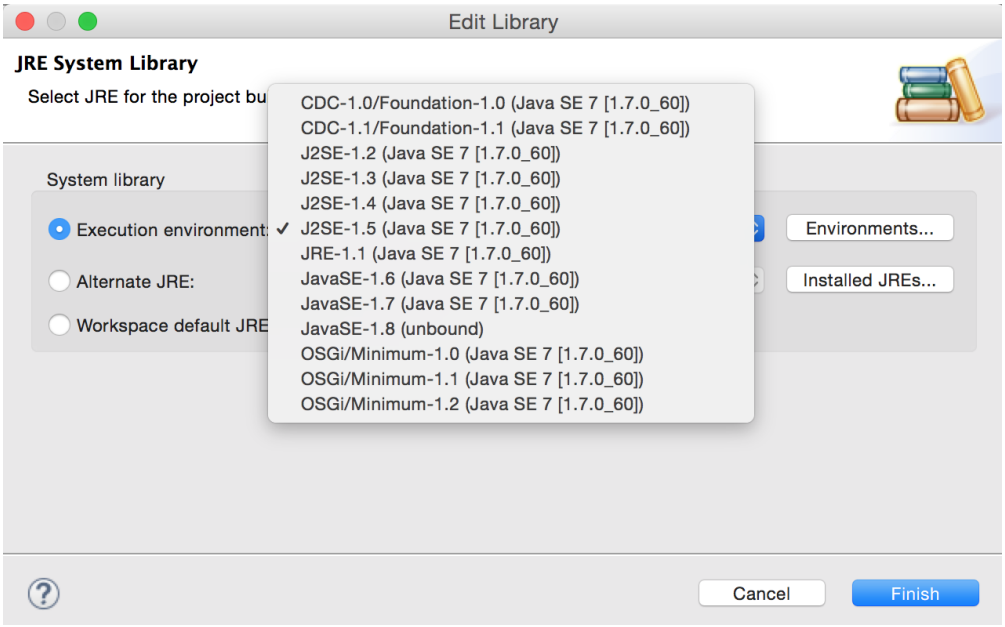
It is possible that the JRE System Library created by Maven is not compatible with the installed JREs. In the screenshot shown below, it is set to **J2SE-1.5**, but the Maven compiler is compatible with **JavaSE-1.7** instead. To change it, right-click on JRE System Library, and select **Build Path** → **Configure Build Path**.



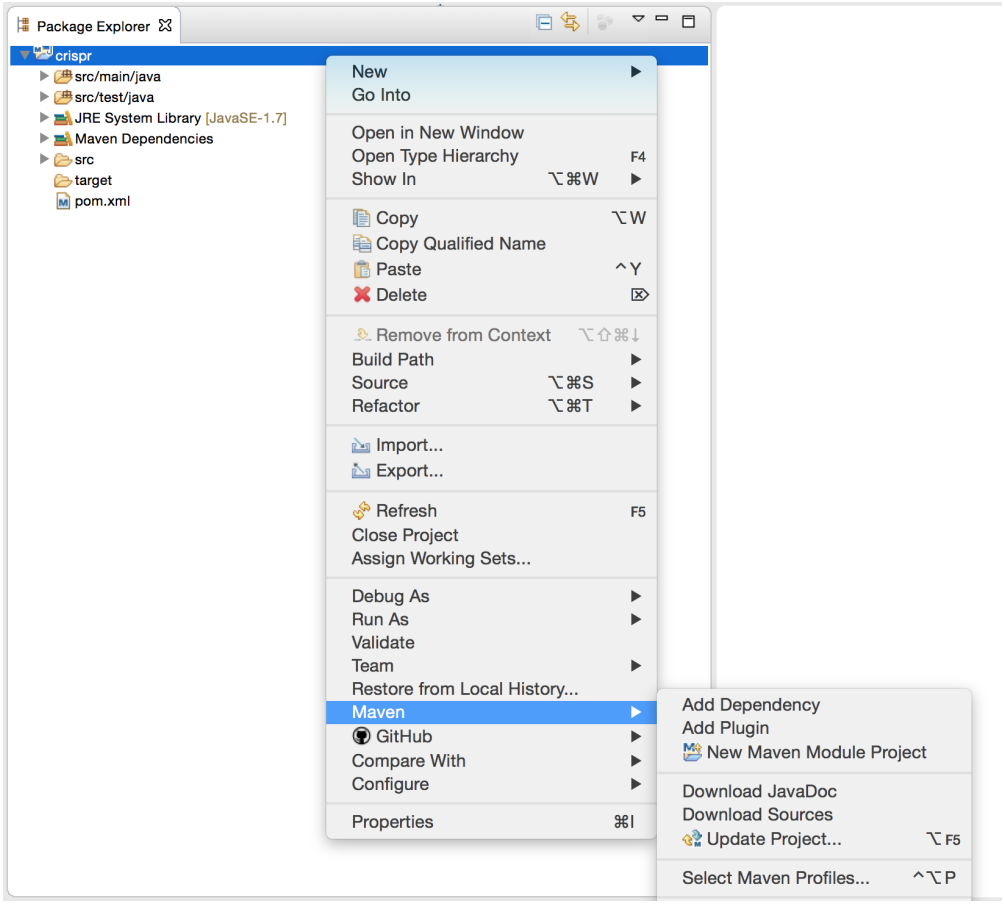
In the popup properties window, select **Edit** under **Libraries**.



We can now change the execution environment to JavaSE-1.7 as shown below.



This solution, however, is temporary. If we right-click on the "crispr" project and then select **Maven** → **Update Project**, the JRE will reset itself back to J2SE-1.5.



A permanent fix would be to manually add the plugin management information below to the pom.xml file.

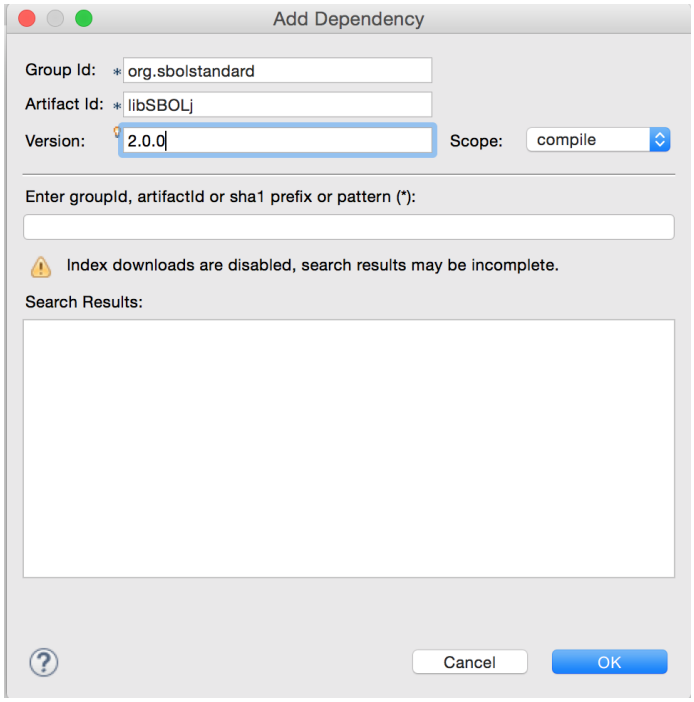
```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

The pom.xml should look like the one shown below after this modification. Remember to save the file and then do **Maven → Update Project**.

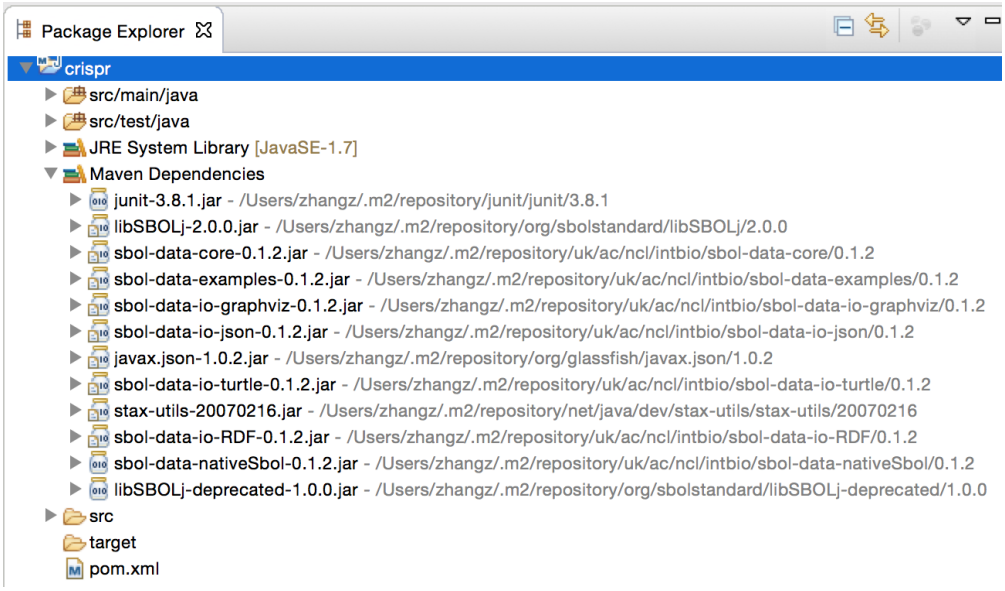
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>mySBOLexamples</groupId>
  <artifactId>crispr</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>crispr</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
</project>
```

Adding `libSBOLj` as dependency

We are now ready to add libSBOLj as a Maven dependency. This can be easily done by right-clicking on the "crispr" project and then select **Maven** → **Add Dependency**. In the popup window shown below, fill in the information for the `libSBOLj` library. The group ID is **org.sbolstandard**, and the artifact ID is **libSBOLj** and the version is **2.0.0**.



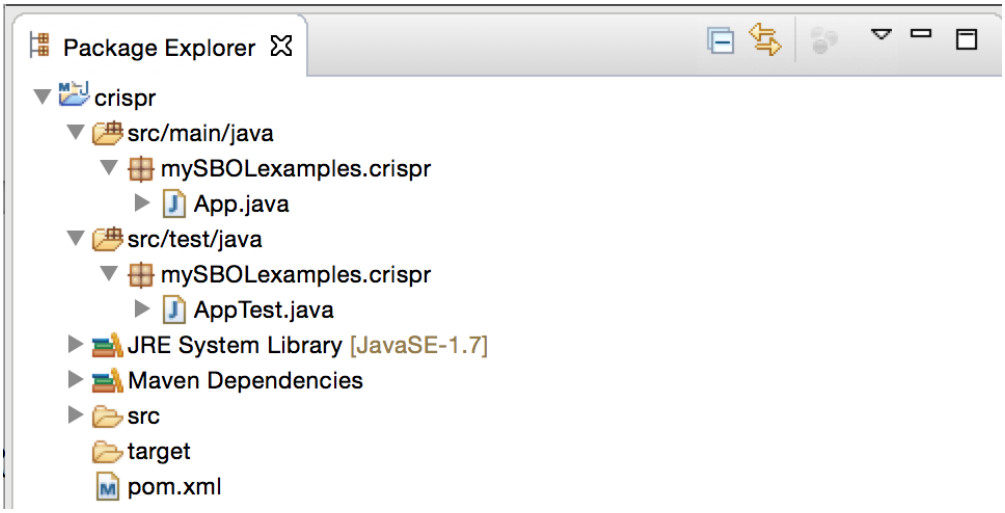
After this dependency is added, Maven automatically brings in the `libSBOLj-2.0.0.jar` and its dependencies from the Maven Central Repository, and places them under the **Maven Dependencies** directory.



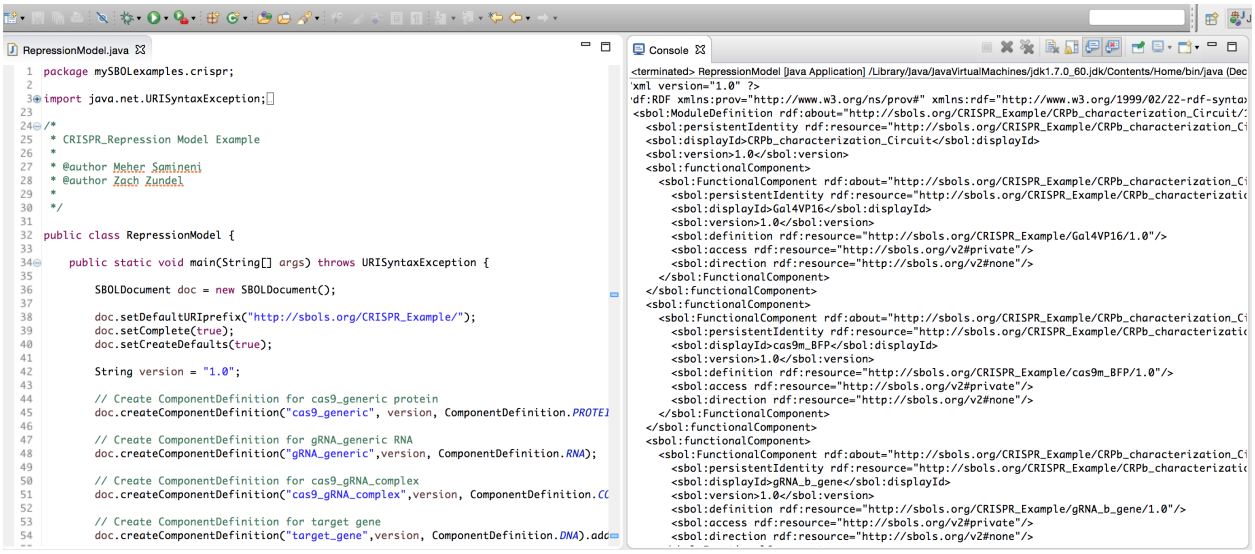
Add the CRISPR repression model

Now we are ready to add the CRISPR model to our project. The model is listed as "RepressionModel.java" under the `libSBOLj` examples directory. When the "crispr" project was created, two template Java class files "App.java" and

"AppTest.java" were automatically created as shown below.



We can modify "App.java" to include the CRISPR model. First, remove all lines in "App.java" except the first line, which contains its package information. Then copy the full contents of "RepressionModel.java" except its package information to "App.java". Lastly, rename "App.java" to "RepressionModel.java". This file should be ready to be executed immediately, and you should be able to see the RDF/XML output in the "console" view. A screenshot is shown below. The "AppTest.java" can be modified accordingly to serve as a test file for the CRISPR repression model.



Modeling CRISPR repression using `libSBOLj 2.0`

We now describe the CRISPR-based repression module and how it can be modeled and encoded using the `libSBOLj 2.0` library. We use bold font in the following text and figure captions to mark available data model in SBOL 2.0.

CRISPR repression model

First, consider the CRISPR-based Repression Template **ModuleDefinition** shown in the center of Figure 1. It provides a generic description of CRISPR-based repression behavior. Namely, it includes generic *Cas9*, *guide RNA* (gRNA), and *target* DNA **FunctionalComponent** instances. It also includes a *genetic production* **Interaction** that expresses a generic target gene product. Finally, it includes a *non-covalent binding* **Interaction** that forms the Cas9/gRNA complex (shown as dashed arrows), which in turn participates in an *inhibition* **Interaction** to repress the target gene product

production (shown with a tee-headed arrow). The CRISPR-based Repression Template is then instantiated to test a particular CRISPR-based repression device, CRPb, by the outer CRPb Characterization Circuit **ModuleDefinition**. This outer characterization circuit includes gene **FunctionalComponents** to produce specific products (i.e., mKate, Gal4VP16, cas9m_BFP, gRNA_b, and EYFP), as well as **FunctionalComponents** for the products themselves. Next, it includes *genetic production* **Interactions** connecting the genes to their products, and it has a *stimulation* **Interaction** that indicates that Gal4VP16 stimulates production of EYFP. Finally, it uses **MapsTo** objects (shown as dashed lines) to connect the generic **FunctionalComponents** in the template to the specific objects in the outer **ModuleDefinition**. For example, the outer module indicates that the target protein is EYFP, while the cas9_gRNA complex is cas9m_BFP_gRNA_b.

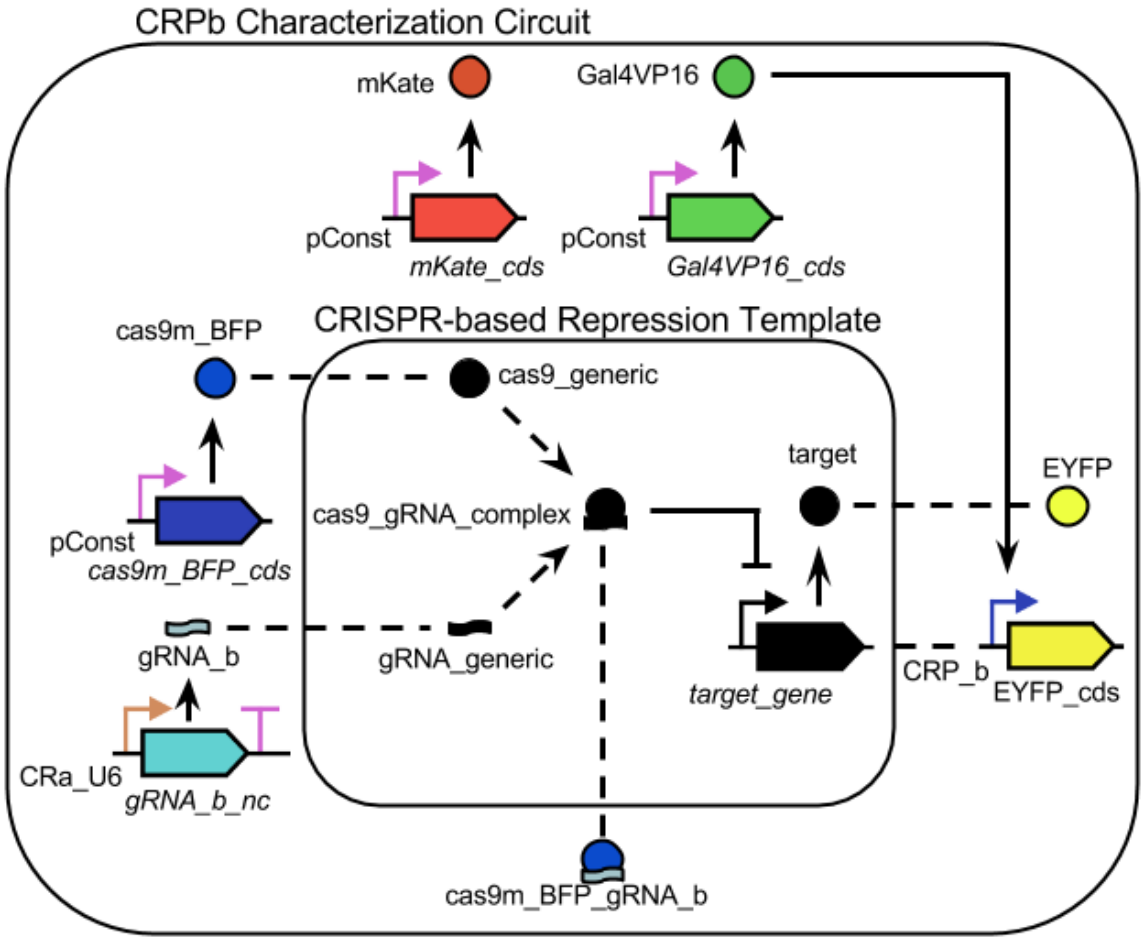


Figure 1: Illustration of a hierarchical CRISPR-based repression module represented in SBOL 2.0 (adapted from Figure 1a in [1]). The CRISPR-based Repression Template **ModuleDefinition** describes a generic CRISPR repression circuit that combines a Cas9 protein with a gRNA to form a complex (represented by the dashed arrows) that represses a target gene (represented by the arrow with the tee arrowhead). These relationships between these **FunctionalComponents** (instances of **ComponentDefinitions**) are represented in SBOL 2.0 using **Interactions**. This **Module** is instantiated in the outer CRPb Characterization Circuit **ModuleDefinition** in order to specify the precise (including **Sequences** when provided) **FunctionalComponents** used for each generic **FunctionalComponent**. The undirected dashed lines going into the template **Module** represent **MapsTo** objects that specify how specific **FunctionalComponents** replace the generic ones.

Encoding using `libSBOLj 2.0`

Program 1 illustrates the use of the `libSBOLj 2.0` library using an excerpt of the Java code to express the CRISPR-based repressor design in SBOL 2.0. First, a new **SBOLDocument** is created (line 1), and is given a default URI prefix (line 2). At this point, **ComponentDefinition** and **Interaction** objects are also created for the CRISPR-based Repression Template **ModuleDefinition** (not shown). Then, **Sequence** objects are created for those sequences provided in [1].[1] For example, to create the sequence for the CRP_b promoter, we call the `createSequence` method with the *displayId* (CRP_b_seq), *version* (1.0), the sequence, and the encoding used (line 3). Note that this method, as well as other create methods in the library, creates a *compliant URI* that has the following form:

http://⟨prefix⟩/⟨displayId⟩/⟨version⟩

using the default URI prefix and provided *displayId* and *version*. The ⟨*prefix*⟩ represents a URI for a namespace (for example, `www.sbols.org/CRISPR_Example`). The author of a **TopLevel** object, such as the **Sequence** object we just created, should use a URI prefix that either they own or an organization of which they are a member owns. When using compliant URIs, the owner of a prefix must ensure that the URI of any unique **TopLevel** object that contains the prefix also contains a unique ⟨*displayId*⟩ or ⟨*version*⟩ portion. Multiple versions of an SBOL object can exist and would have compliant URIs that contain identical prefixes and displayIds, but each of these URIs would need to end with a unique version. Lastly, the compliant URI of a non-**TopLevel** object is identical to that of its parent object, except that its displayId is inserted between its parent's displayId and version. This form of compliant URIs is chosen to be easy to read, facilitate debugging, and support a more efficient means of looking up objects and checking URI uniqueness.

Program 1: Fragments of Java code to produce part of the CRISPR Repression example using `libSBOLj 2.0`.

```
1  SBOLDocument doc = new SBOLDocument();
2  doc.setDefaultURIprefix("http://sbols.org/CRISPR_Example/");
3  doc.createSequence("CRP_b_seq", "1.0", "GCTCCGAATTTCTCGACAGATCTCATGTGAT...", Sequence.IUPAC_DNA);
4  ComponentDefinition CRP_b = doc.createComponentDefinition("CRP_b", "1.0", ComponentDefinition.DNA
       );
5  CRP_b.addRole(SequenceOntology.PROMOTER);
6  CRP_b.addSequence("CRP_b_seq");
7  doc.createComponentDefinition("EYFP_cds", "1.0", ComponentDefinition.DNA).addRole(
       SequenceOntology.CDS);
8  ComponentDefinition EYFP_gene = doc.createComponentDefinition("EYFP_gene", "1.0",
       ComponentDefinition.DNA);
9  EYFP_gene.createSequenceConstraint("EYFP_gene_constraint", RestrictionType.PRECEDES, "CRP_b", "
       EYFP_cds");
10 doc.createComponentDefinition("Gal4VP16", "1.0", ComponentDefinition.PROTEIN);
11 ModuleDefinition CRPb_circuit = doc.createModuleDefinition("CRPb_characterization_circuit", "1.0"
       );
12 Interaction EYFP_Activation = CRPb_circuit.createInteraction("EYFP_Activation",
13                                                             SystemsBiologyOntology.STIMULATION);
14 EYFP_Activation.createParticipation("GAL4VP16", "Gal4VP16").addRole(SystemsBiologyOntology.
       STIMULATOR);
15 EYFP_Activation.createParticipation("EYFP_gene", "EYFP_gene").addRole(SystemsBiologyOntology.
       PROMOTER);
16 Module Template_Module = CRPb_circuit.createModule("CRISPR_Template", "CRISPR_Template", "1.0");
17 Template_Module.createMapsTo("EYFP_gene_map", RefinementType.USELOCAL, "EYFP_gene", "target_gene"
       );
```

Next, we create **ComponentDefinition** objects for each element in the module. For example, a **ComponentDefinition** of DNA type is created for the CRP_b promoter (lines 4-6). Note that by using compliant URIs, the sequence can be looked up using its *displayId*, and since no version is provided, it is referenced by its *persistentIdentity* URI (line 6). It is simply its URI without the version when using compliant URIs. The purpose of a persistent identity is to allow an object to refer to the latest version of another object using this URI. The latest version of an object is determined using *semantic versioning* conventions (c.f., `http://semver.org/`).

[1] Unfortunately, as usual, not all sequences are provided in the paper.

Next, we create two **ComponentDefinition** objects, one for the EYFP *coding sequence* (CDS), and one for the EYFP gene (lines 7-8). We use a **SequenceConstraint** object (line 9) to indicate that the CRP_b promoter precedes the EYFP CDS, because the sequence for the CDS has not been provided and thus cannot be given an exact **Range**. Finally, we create a protein type **ComponentDefinition** for the Gal4VP16 protein (line 10). After all the **ComponentDefinitions** are created, we create a **ModuleDefinition** object for the CRPb Characterization Circuit (line 11).

Next, the **Interactions** between the components are specified using terms from the *Systems Biology Ontology* (SBO) [2]. One example **Interaction** is the stimulation of the `EYFP_gene` by the Gal4VP16 protein (lines 12-14). Now, the CRISPR-based Repression Template **Module** is instantiated and connected to the CRPb Characterization Circuit using **MapsTo** objects. For example, a **MapsTo** object is used to indicate that the `target_gene` in the template should be refined to be the `EYFP_gene` specified in the CRPb circuit (line 17).

As we mentioned previously, the complete repression model is described in the "RepressionModel.java" under the libSBOLj examples directory. This example is self-contained in that you can run it to generate the RDF/XML output. Also, SBOL does not provide the specification of a mathematical model directly. It is possible, however, to generate a mathematical model using SBML [3] and the procedure described in [4]. Then, the SBOL document can reference this generated SBML model.

References

[1] S. Kiani, J. Beal, M. Ebrahimkhani, J. Huh, R. Hall, Z. Xie, Y. Li, and R. Weiss, "Crispr transcriptional repression devices and layered circuits in mammalian cells," *Nature Methods*, vol. 11, no. 7, pp. 723–726, 2014.

[2] M. Courtot, N. Juty, C. Knüpfer, D. Waltemath, A. Zhukova, A. Dräger, M. Dumontier, A. Finney, M. Golebiewski, J. Hastings, S. Hoops, S. Keating, D. Kell, S. Kerrien, J. Lawson, A. Lister, J. Lu, R. Machne, P. Mendes, M. Pocock, N. Rodriguez, A. Villeger, D. Wilkinson, S. Wimalaratne, C. Laibe, M. Hucka, and N. Le Novère, "Controlled vocabularies and semantics in systems biology," *Molecular Systems Biology*, vol. 7, 2011.

[3] M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, H. Kitano, A. Arkin, B. Bornstein, D. Bray, A. Cornish-Bowden, *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[4] N. Roehner, Z. Zhang, T. Nguyen, and C. Myers, "Generating systems biology markup language models from the synthetic biology open language," *ACS Synthetic Biology*, vol. 4, no. 8, pp. 873–879, 2015.