Domain-Specific Modeling and Domain Driven Development using UDM Tutorial

Attila Vizhanyo, email:viza@isis.vanderbilt.edu
Sandeep Neema, email: Sandeep.Neema@vanderbilt.edu
Nora Somogyi, email: Nora.Somogyi@vanderbilt.edu
Institute for Software Integrated Systems, Vanderbilt University

This tutorial describes the *domain-specific modeling (DSM)* approach, and explains how to create and use *domain-specific modeling languages (DSML)* to identify and capture the significant concepts of a *domain* and their relationships and attributes. The DSM approach is targeted at application developers who have extensive domain knowledge as well as firm foundations in their professions, but may not be formally trained in computer science.

Modeling is a central part of all the activities that lead up to the deployment of good software. In general, *models* provide just a simplified, schematic description of a particular complex system or process (commonly referred as a *domain*) to promote the intelligibility and integrity of the domain. We build models to communicate the desired structure and behavior of our system, to visualize and to better understand the system we are building. Domain-specific modeling languages let you identify what models of a specific domain can consist of, along with the domain properties and its static structure. For example, some molecular modeling languages specialize on molecular mechanics or dynamics only.

Software engineers and computer scientists realized long ago that by designing a language that is specialized to model your domain of interest has many benefits.

1. In DSM, the models are constructed using concepts that represent a given application domain, not concepts of a programming language. The modeling language follows the domain *abstractions* and *semantics* (semantics def: the *meaning* of the domain concepts, their relation and properties). Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. The final products are generated automatically from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only one domain. Your expert defines them, your developers use them.
2. DSMLs encourage the use of Domain Driven Development techniques, which accelerate software development, and allow for rapid prototyping and implementation of domain-specific systems. More on this later.
3. The modeling language also generates a precise *type system*, which will ensure the *syntactical correctness* of your models via *data structuring* and *data types*. Strong type systems guarantee correctly formatted models by construction. Further *semantic rules* can be enforced by introducing so-called *constraints*, that will prevent users from creating syntactically valid but illogical or meaningless data.
4. DSMLs act as schemas for data that needs to be exchanged. E.g. XML Schemas (XSDs) provide a way for defining the structure and semantics of XML documents. Schemas, in general, let independent tool vendors understand-, and agree on common data interchange formats. Such an agreement becomes a hard-to-violate

formal contract with the introduction of DSM. Closely related is the use of DSMLs for data that needs to be stored, possibly in a repository.

5. One of the strongest advantages of applying the DSM approach lies in *domain evolution,* i.e. maintenance and migration. There is never, ever, going to be the definitive ultimate model that describes your domain perfectly, hence there is never going to be a perfect DSML. Models are successive approximations, you will always find new concepts or will want to change existing ones in order to get a better representation of your domain. Domain evolution incurs a significant development overhead in tool maintenance, and we will explain later how this overhead can be reduced or eliminated with Domain Driven Development tools.

It should be clear by now that designing such a DSML is an activity, which is typically performed by domain experts. This design activity is commonly known as *metamodeling*, and it produces, among other things, *metamodels*. Metamodeling is a close relative to modeling, such as object-oriented modeling, or biological modeling in that it produces models (metamodels) which are an attempt at describing the world around us for a particular purpose. Metamodels specify precisely the modeling paradigm, i.e. the concepts and notation the end user will build his model with. The experience and intuition of the expert, combined with domain rules are the real sources of clues. Metamodels can be specified either in visual or textual form. An UML class diagram is an industry standard, simple and powerful visual modeling language that is designed to model the static design view of object oriented systems. UML class diagrams are also suitable for describing domains, or equivalently, metamodels. The easiest way to understand UML class diagrams is to try an example.

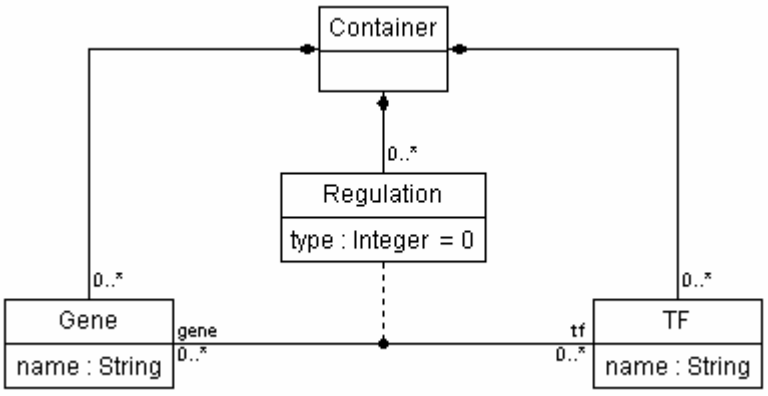

Figure 1 GeneTF UML class diagram

In this example we want to create a domain that can model how genes interact with transcription factors in a biochemical system. We have just identified two significant concepts in our domain, i.e. genes and transcription factors. That is, our models will consist of "things" that are either genes or transcription factors. We call these things that constitute the model *objects*. Objects of a model are unique instances of concepts of a domain. Instead of saying "is-a", we rather say that an object is instance of a particular *type*, or equivalently, of a *class*. A good analogy is the classification of creatures into species. The fundamental advantage of classification is that we can identify a set of related things sharing some common property, separate them from others, and can reason

about them in a uniform manner. In this example, we classified the objects of a model into genes and transcription factors, that is, we created two types (classes): Gene and Transcription Factor (TF). Gene and TF instances are the primary entities of a GeneTF model. The class diagram in Figure 1 describes the same with a different notation. There are other elements in this diagram, but focus only on the Gene and TF rectangles. The rectangle is the UML graphical representation of a class. The class specification consists of a class name, and the attribute(s). The *class name* distinguishes the class from other classes. The class name is a unique textual string. An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. In this example, both Gene and TF has an attribute called "name", which is used to give unique names to Gene or TF objects. (Don't mix the class name with the name attribute. The former is the name of the class, the later is an attribute of objects of that class). So we can see how attributes specify properties of domain concepts. The domain specification is not yet complete, because we have not specified the relationships our objects can establish in a GeneTF model. A *relationship* is a connection among things. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships. We often want to structure our model, so that specific objects are limited to exist inside well-defined places, or *containers*. We say that the container object is the parent of the object, the contained object is the child, and this parent-child relationship is expressed using a *composite* relationship. The UML notation for composition is a connection between the parent and the child with a black diamond at the parent end. What the UML class diagram says in Figure 1, is that Gene and TF objects must reside in Container objects. The Container class is not really a domain concept, that is, it does not have physical counterpart in real GeneTF systems. The Container type is rather just an abstraction, it merely represents a parent type for the classes Gene and TF. Another UML structural relationship is *association*. Association specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. We started creating the GeneTF domain with the promise of that the domain will be suitable for expressing gene-transcription factor interactions. More precisely what we need to model is that which genes can be transcripted with which TFs, and how. These interactions are best modeled with the help of associations. We want to connect those Gene-TF object pairs that can interact together. This association is represented with a three-legged relationship in the UML class diagram. We already know two types, and their role in the association: Gene and TF. The third type is called Regulation and it is a specific class, called association class. An *association class* describes the name and the attributes of the association, it is connected to. Association classes are connected to associations with dashed lines. The name of this association is Regulation, and it has an attribute called type. Type is an integer number, with specific codes interpreted as regulation types, that describe *how* genes and transcription factors interact. So we see that the type of the interaction is specified by association attributes. This completes the discussion of the UML class diagram in Figure 1.

So now that we have described the semantics of the UML class diagram, let us continue with the description of the Domain Driven Development process. We have created a

DSML, now our goal is to use it to read and manipulate domain specific models as easily and safely as possible with the least amount of work needed. How can we achieve these seemingly contradictory design objectives at the same time? The answer is we don't. We let *metaprogrammable* generation tools to do the work for us.

In practice, the most widespread method of reading and manipulating models is using general purpose programming languages, like C++, Java or Python, even today. This is so mainly because these model operations can be greatly diverse and ultimately are executed as low-level machine instructions in a digital computer. The languages mentioned above have rich semantics, that make them suitable for most applications. Also, the semantics of these languages are close to a machine understandable representation.

To leverage the benefits of the domain specifications (which are UML class diagrams in our case), we need a *bridge*, i.e. a translator tool, which would create some special program code out of the domain specifications. This program code provides the same representation of the domain as UML class diagrams, but written in a general purpose programming language, like C++, Java or Python. We call this program code *Domain-Specific Application Program Interface (Domain-Specific API).* The application developer, who creates, reads or writes models, will be able to use this generated Domain-Specific API to perform domain-specific operations on models. The emphasis is on *domain-specific operations*, that is the developer can only perform operations that are specified in the domain specification. He/she can not possibly break the rules established by the domain experts. This is of great importance in achieving data syntactical correctness and (semantic) integrity. Moreover many application developers can use the same generated API again and again, because the API is implemented as a well-encapsulated reusable component. This promotes tool interoperability via ensuring structural and type integrity of data between the tools.

So we see the benefits of using DSMLs to generate a Domain-Specific APIs in a general-purpose programming language. But why do we need modeling at all, why don't we create the Domain-Specific API from scratch, manually? Well, for several good reasons. First of all, take a look at Figure 2, which shows the Domain-specific API generated for class Gene in Java. This code is in one-to-one correspondence to the Gene UML class definition and its relationships in Figure 1. As a domain designer, which representation would you choose to specify your domain? Visual languages along with a corresponding generator tools allows for rapid prototyping and implementation, what domain and language designers are so enthusiastic about. This is so because the UML class diagram semantics are hidden and tailored for the specification of static design view of systems or domains. Java on the other hand is much more general language, and you must be explicit if you want to express something specific, such as system design. Second, UML class diagrams are much easier to understand and use than Java. Domain experts are not necessarily professional software engineers who know the inside-out of complex general purpose programming languages, so they will unlikely to come up with more robust, efficient implementations than dedicated and optimized generator tools can create. Take a test how much time you need to understand class Gene in UML notation and in the Java code. UML class diagrams help others (or maybe yourself after some months) to understand domains very quickly and intuitively. Better yet, the two representations

```
package edu.vanderbilt.isis.genetf.genetf;
import edu.vanderbilt.isis.udm.*;

/**
 * Domain specific class of <code>Gene</code>.
 */
public class Gene extends UdmPseudoObject
{
	// meta information
	public static final String META_TYPE = "Gene";
	public static final String META_TYPE_NS = "GeneTF";
	private static UdmPseudoObject metaClass;

	/**
	 * Constructor.
	 * @param  upo The object that helps the initialization of the instance
	 * @param  metaDiagram The diagram of the data network
	 * @throws  UdmException If any Udm related exception occured
	 */
	protected Gene(UdmPseudoObject upo, Diagram metaDiagram)
		throws UdmException
	{
		super(upo, metaDiagram);
	}

	/**
	 * Returns the meta class.
	 * @return  The meta class
	 */
	UdmPseudoObject getMetaClass()
	{
		return metaClass;
	}

	/**
	 * Creates an instance of the class in the container specified by
	 * the parameter.
	 * @param  parent The parent container
	 * @return  An instance of the class <code>Gene</code>
	 * @throws  UdmException If any Udm related exception occured
	 */
	public static Gene create(Container parent)
		throws UdmException
	{
		Diagram metaDiagram = parent.getDiagram();
		return new Gene(parent.createObject(META_TYPE, META_TYPE_NS),
			metaDiagram);
	}
	/**
	  * Attribute for <code>name</code>.
	  */
	public static final String name = "name";

	/**
	 * Sets the value of the attribute <code>name</code> to a value
	 * specified by the parameter.
	 * @param _v The new value of the attribute
	 * @throws  UdmException If any Udm related exception occured
	 */
	public void setname(String _v)
		throws UdmException
	{
```

```
            setStringVal(name, _v);
        }

        /**
         * Returns the value of the attribute <code>name</code>.
         * @return  The value
         * @throws  UdmException If any Udm related exception occured
         */
        public String getname()
            throws UdmException
        {
            return getStringVal(name);
        }

        /**
         * Sets the other ends of the association with role name <code>tf</code>.
         * @a The other ends of the association
         * @throws  UdmException If any Udm related exception occured
         */
        public void settf(Regulation[] a)
            throws UdmException
        {
            setAssociation("tf", new UdmPseudoObjectContainer(a),
                UdmHelper.CLASS_FROM_TARGET);
        }

        /**
         * Returns the other ends of the association with role name
         * <code>tf</code>.
         * @return The other ends of the association
         * @throws  UdmException If any Udm related exception occured
         */
        public Regulation[] gettf()
            throws UdmException
        {
            UdmPseudoObjectContainer objs = getAssociation("tf",
                UdmHelper.CLASS_FROM_TARGET);
            return (Regulation[]) Utils.wrapWithSubclass(objs,
                Regulation.class, getDiagram());
        }
}
```

Figure 2 Gene.java: Domain-Specific Java API for class Gene

are equivalent. If you see a UML class definition in a UML class diagram, you will be confident that you can have access to all attributes, relationships exactly the same way using the Domain-Specific API.

Finally think long term, and the innumerous ways your domain can evolve. You might need finer grained control over the specification of transition factors, or you want to subcategorize genes based on some new gene properties that was not relevant at the initial design of the domain. As your domain evolves and gets bigger, it will get more complex too, you will need just to reorganize, or *refactor* your domain without modifying its semantics, and so on. Domain evolution is an inevitable process that leaves behind many dependent models, tools and other artifacts obsolete and invalid. Automatic generation tools are not panacea to this problem, but they certainly reduce the burden of tool upgrade and maintenance. The Domain-Specific API can be regenerated automatically from UML class diagrams, anytime, without any user intervention. The upgraded Domain-Specific API may break existing legacy application code (it depends

on the nature of change performed on the domain), but since the user code is using domain abstractions, the amount of work needed should be small. The last problem is upgrading, or transforming existing models so they reflect the changes of the new, modified domain. We also provide a formal language for the specification of such model transformers, and a tool suite that executes the model transformers on those existing models, and perform the same transformation steps consistently. So you can see that DSM techniques will also help you in reducing development time needed for maintenance as your system evolves.

The *Universal Data Model (UDM)* framework, developed at the Institute for Software Integrated Systems, Vanderbilt University, can assist you in the introduction of Domain Specific Modeling techniques into your existing systems. The UDM framework includes the development process and set of supporting tools that are used to generate Java or C++ programmatic interfaces from UML class diagrams. These interfaces and the underlying libraries provide convenient programmatic access and automatically configured persistence services for data structures as described in the input UML diagram. The UDM development process is shown in Figure 3. Starting from the left, the input of the process is the domain specification, i.e. the metamodel, that describes the domain of interest. The representation is in XML format, and it must be valid according to the UML XML Schema. The UDM code generator tool (Udm.exe) generates the following files out of the domain specification.

- *.java* or *.cpp, .h* files: Domain-Specific API is supported in C++ or Java
- *.xsd* file: XML Schema which can be used to evaluate the domain models.

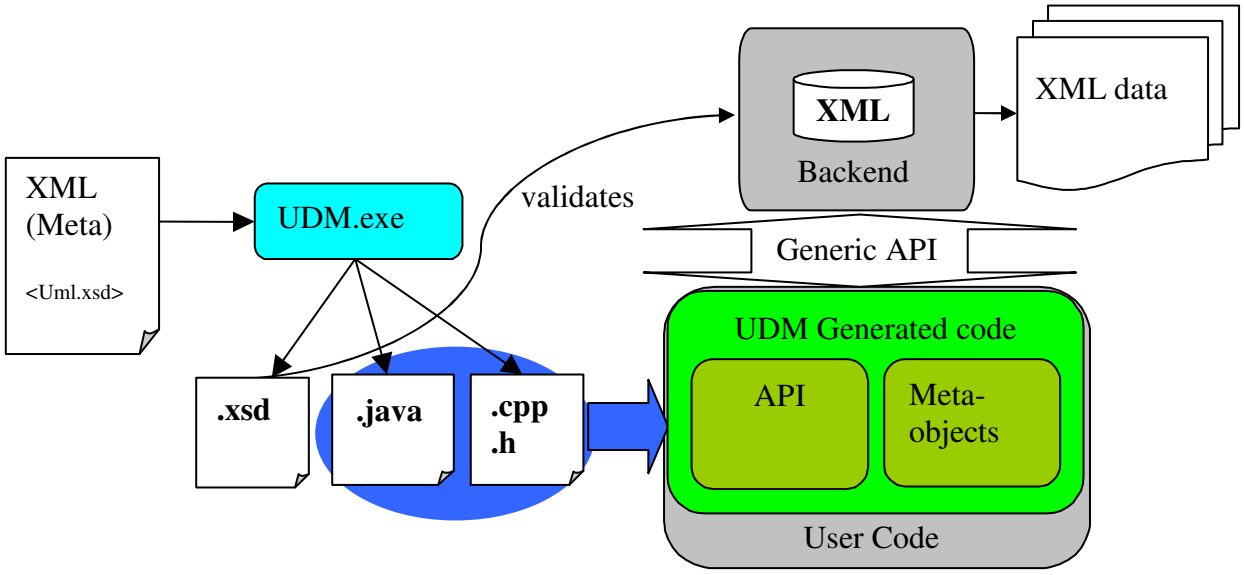

Figure 3 The UDM Framework

The generated Domain-Specific API then can be used from the User Code to access the domain specific models. As Figure 3 shows, the Domain-Specific API generated by the

UDM code generator tool, along with the UDM Generic Framework together form a *middle layer* between the User Code and the domain specific models. This middle layer hides low-level implementation details that are not relevant from the domain user's perspective. Instead of bothering with machine-level abstractions, the user can concentrate on the domain problem using high-level domain concepts exclusively. The result is a cleaner, safer, better organized User Code, which is more accessible to the domain users.

In the following section, we explain the generated Domain-Specific API in Java, its relation to the UML class diagram that specifies the domain, and discuss some of its usage scenarios. There is a Java package generated for each UML class diagram, a Java package generated for each namespace, and there is a Java public class definition generated for each UML class in the class diagram in the appropriate package. Please refer to Figure 2 for the following discussion. Gene.java starts with a package definition which corresponds to the GeneTF diagram and namespace: *package edu.vanderbilt.isis.genetf.genetf;*. Since no namespace has been defined for the package, the namespace of the Gene.java class is the default namespace, i.e., the name of the diagram. Next, the udm general framework package gets imported: *import edu.vanderbilt.isis.udm.*;*. *UdmPseudoObject* is the common base class for every type defined in any domain, which is expressed for class Gene in Java: *public class Gene extends UdmPseudoObject.* You can create Gene objects in your model using the static *create()* method. *Create()* takes a *parent* object as its argument, which specifies the container in which the Gene object should be created. If you remember the GeneTF class diagram, Gene-s are contained in Container-s, so the parent type is specified as *Container* in the generated Java function signature, accordingly. After *Create(),* you find the attribute setter/getter functions in the *Gene* class definition. For each UML class attribute, a pair of functions is generated to set and get the value of the attribute. The *setname()* method sets the name attribute of a gene object to the string contained in the *_v* argument, while *getname()* returns the current value of the name attribute. This pattern of set/get<Attribute Name> repeats itself for each class attribute in the generated Domain-Specific API. Next we examine how associations can be queried, created and deleted. Genes can be associated with transition factors through regulation association classes. This three-legged association is represented with two two-legged associations in the Domain-Specific API: one association between Gene and Regulation, and another one between Regulation and TF. Because these associations can be traversed from both directions, associations can be queried or created starting from either end of the associations. For example, Gene-Regulation associations can be queried using *Gene.gettf()* or *Regulation.getgene()*. The code snippet in Figure 4 creates an association between a Gene object and a TF object by using the association setter methods of a Regulation object. *Con* is the parent *Container* object created earlier. This code also demonstrates how to use the attribute setter methods.

```
protected void createGeneTFAssoc(Container con) throws UdmException {
       Gene g1 = Gene.create(con);
       g1.setname("gene1");
       TF t1 = TF.create(con);
       t1.setname("tf1");
       Regulation r1 = Regulation.create(con);
       r1.setgene(g1);
       r1.settf(t1);
       r1.settype(2);
}
```

Figure 4 User Code that creates a Gene-TF association

```
protected void printContainer(Container con) throws UdmException {
       Gene[] genes = con.getGeneChildren();
       for(int i=0; i<genes.length; i++)
       {
            Gene g = genes[i];
            String n = g.getname();
            System.out.print("Gene " + n + " regulates: " );
            Regulation[] regs = g.gettf();
            for(int j=0; j<regs.length; j++)
            {
               try {
                   if (j > 0) System.out.print(", ");
                   Regulation r = regs[j];
                   TF t = r.gettf();
                   String tn = t.getname();
                   System.out.print(tn);
               }
               catch (UdmException e) {
                   System.out.print("Gene " + n + "fails " );
                   throw e;
               }
            }
            System.out.println();
       }
  }
```

Figure 5 User Code that queries and prints Gene-s and associated TF-s

Figure 5 also demonstrates how intuitive and easy it is to use the generated Domain-Specific API to query data in domain-specific models. This code will print out all the genes in *con*, along with the set of transition factors the actual gene regulates. Udm uses the exception mechanism for error handling, which means you must catch *UdmException*-s in your applications for safe operation. However, we have not been talking about how the model itself can be accessed using the UDM framework. This is what we discuss next.

The UDM framework supports multiple backends for the representation of domain-specific models, but we are going to focus exclusively on the XML backend for the sake of simplicity. XML is an standardized language suitable for describing structured data. The UDM framework uses XML to store the models persistently, and uses XSD to validate the correctness of XML documents. The UDM framework provides programmatic access to the (meta-)model files through the *UDM generic API*. The UDM generic API introduces domain-independent abstractions such as *data networks* and *objects.* Data networks represent models, and they are containers consisting of model *objects*. You can open existing data networks from files, streams, or strings, or create new ones from scratch in files, streams, or strings, as shown in Figure 6.

```
// manipulate data network via file and stream
ContainerFileFactory gtf =
     FactoryRepository.getGeneTFContainerFileFactory();
// create new datanetwork
Container con = gtf.create("GeneTF-newmodel.xml");
// open existing datanetwork from file
//Container con = gtf.open("GeneTF-existingmodel.xml");
// open existing datanetwork from stream
// InputStream stream = …;
//Container con = gtf.open(stream);
createGeneTFAssoc( con);
printContainer( con);
// call constraint checker
String res = gtf.checkConstraints();
System.out.println("Result of constraint evaluation: " + res);
// save & close
gtf.save();
// close without saving
//gtf.close();
// save in a new file & close
//gtf.saveAs("GeneTF-newmodel.xml");
// save in a stream & close
//InputStream result = gtf.saveAsStream();

// manipulate data network via string and stream
ContainerStringFactory gts =
           FactoryRepository.getGeneTFContainerStringFactory();
// create new datanetwork in string
Container con = gts.create( );
// open existing datanetwork from string
// String xmlString = …;
//Container con = gts.open(xmlString);
// open existing datanetwork from stream
// InputStream stream = …;
//Container con = gts.open(stream);
createGeneTFAssoc( con);
printContainer( con);
// call constraint checker
String res = gts.checkConstraints();
System.out.println("Result of constraint evaluation: " + res);
// save & close
String new_dn = gts.save();
// close without saving
//gts.close();
// save in a stream & close
//InputStream result = gts.saveAsStream();
```

Figure 6 Using UDM data networks

Class *FactoryRepository*, *ContainerFileFactory*, and *ContainerStringFactory* are domain-specific data network factories generated automatically by the UDM code generator tool. As a domain user, you can create new– or open existing GeneTF data networks (i.e. domain-specific models) using the services these factories provide. Class *FactoryRepository* is a collection of namespace and root object specific data network factories and contains static method to access these factories. The UDM code generator tool generates two different types of data network factories for each *root object* in the data network as their name show: one for manipulating data networks via file and stream operations and one for manipulating data networks via string and stream operations. The

factories provide common interfaces to manipulate the data network, such as *open()*, *create()*, *save()*, *close()*, and *checkConstraints()*.
In case of *ContainerFileFactory* both *create()* and *open()* take the file names of the models as their arguments, and both return the *root object* of the created/opened data network. In addition, the *open()* can take an *InputStream* as a parameter and also returns the *root object* of the opened data network. The root object is the topmost element of the object hierarchy inside a model, which is automatically accessible from the data network. The type of the root object of GeneTF models is *Container*, as you can check in the class diagram in Figure 1, consequently the return type of these methods is *Container*. With a *Container* object in your hands you can start building or reading GeneTF models in a hierarchical fashion, and you can also evaluate constraints of the data network. After modifying the data network, you can save the changes and store the modified data network in a file or in a stream by calling the appropriate *save()* function. You can choose not to save the changes of the data network and call *close()*.
Similarly to *ContainerFileFactory*, *open()*in the *ContainerStringFactory* takes the model as a string as its argument and returns the *root object* of the opened data network. In addition, the *open()* can take an *InputStream* as a parameter and also returns the *root object* of the opened data network. However, *create()* does not take any parameter as its argument because the created data network is stored in a string in the memory, but it also returns the *root object* of the created data network. After modifying the data network, you can save the changes and store the modified data network in a string or in a stream by calling the appropriate *save()* function. You can choose not to save the changes of the data network and call *close()*.
Figure 7 shows the GeneTF model after executing the first code snippet in Figure 6. This XML document contains a Container element, which contains three other elements: a Gene, a Regulation and a TF. Observe that their attributes (name, type) are set accordingly to the user code in Figure 4. The associations are represented with unique identifiers: element Gene is associated with an element whose identifier is “id2d”, which is element Regulation, and so on. The XML format is designed to be human-readable, nevertheless if you find this representation bulky and hard-to-read, you can implement domain-specific viewers and editors that would visualize your domain-specific data in a more comprehensible format. It is important to note that you could reuse the generic UDM framework and the Domain-Specific API for parsing the data, and can concentrate on the viewer and editor features without worrying about data representation issues. And you can reuse the Domain-Specific API in many other applications, even use multiple Domain-Specific APIs in one application to manipulate various domains. The benefit of using DSM concepts and techniques becomes quickly manifold.

```
<GeneTF:Container xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:GeneTF="http://www.isis.vanderbilt.edu/2004/schemas/GeneTF"
xsi:schemaLocation="http://www.isis.vanderbilt.edu/2004/schemas/GeneTF
GeneTF.xsd">
       <GeneTF:Gene tf="id2d" _id="id29" name="gene1"/>
       <GeneTF:Regulation _id="id2d" type="2" tf_end_="id1c"
gene_end_="id29"/>
       <GeneTF:TF _id="id1c" gene="id2d" name="tf1"/>
</GeneTF:Container>
```

Figure 7 GeneTF model in XML format