A EDMA tutorial

This tutorial will be loosely based on the Diving School example in the report. If you have not already read through this example, now would be a good time to at least skim through it.

To follow this tutorial you need IntelliJ (or another Java Development Environment) the EDMA_Runtime.jar and The EDMA_Generator.jar file. In this tutorial we will take you through all the steps that are required to build a working example of the Diving School Course Registration System with EDMA. This includes the following steps:

- Set up a new EDMA project in IntelliJ.
- Define global value domains.
- Define the data model for the course registration system.
- Define the actions and views for the course registration system.
- Implement the actions and views.
- Connect to the runtime and use the course registration system.

A.1 Set up a new EDMA project

1. Create a new java project in IntelliJ named *DivingSchool*.
2. Right click the main directory in project explorer and choose *new* → Directory. Write *lib* as the name of the directory and click *finish*.
3. Copy EDMA_Runtime.jar to the newly created lib directory.
4. Right click the project in the package explorer and choose *properties*. Now click on *Java Build Path* in the left part of the window. In the right part of the window click on the tab *Libraries* and then on the button *Add JARs...* Navigate to the *lib* folder and choose the EDMA.jar file. Click *OK* in both windows.
5. Right click on the project in package explorer and choose *new* → *Folder*. Write *edma* as the name of the folder and click *finish*. Right click on the newly created edma folder and choose *new* → *File*. Write Common.edma as the file name and click *finish*. Right click the new Common.edma file and choose *Open with* → *Text Editor*. Now write this line exactly:

   ```
   ValueDomain Name : String[1..MAX]
   ```

   Make sure you write exactly the above line and then press *Ctrl-S* to save the file. You have now created your first global value domain called Name. The values in Name are strings that are at least 1 character long. You will learn more about how to create value domains in a moment.

6. Right click on the project in package explorer and choose *new* → *Class*. Write Make as the name of the class and click finish. Write this code into the class:

```
import edma.generator.EdmaGenerator;

public class Make
{
  public static void main(String[] args)
  {
    //This will be the name of the environment
    String environmentName = "DivingSchool";
    //This is the path to the folder with your EDMA files
    String edmaSrcDir = "C:/Workspace/DivingSchool/edma";
    //This is the path to the java src folder
    String javaSrcDir = "C:/Workspace/DivingSchool/src";
    //This is the root package of the generated code
    String rootPackage = "tutorial";
    new EdmaGenerator(environmentName,
              edmaSrcDir,
              javaSrcDir,
              rootPackage).make();
  }
}
```

   Make sure you replace the paths with the actual paths on your machine. Now right click on this class in the package explorer and choose *Run As* → *Java Application*. The output should look like this:

```
Gathering and compiling input files...
Generating environment generator...
Generating value domain classes...
Deleting generated directories...
Writing generated java files...
All done!
```

   Now right click the project in package explorer and choose *refresh*. You should now see some generated packages in the project. Every time you make any changes to your EDMA files you must run this Make class and refresh the project.

When you run the Make class it creates a new instance of the EdmaGenerator class and calls the make() method on it. The compiler now looks for files in the edma folder and its sub folders that has the extension ".edma". All these files are then compiled to a meta model and the generator then uses this meta model to generate all the java interfaces and classes. If you look at the generated packages and java files, you should see a package called *generated* as a sub package of *tutorial.divingschool* (depending on the names you gave for the project and root package). This package and everything inside it will be deleted and re-generated whenever you call the make() method on the generator class, so you should never put anything into this package or make changes to the files inside this package or any of its sub packages.

A.2 Define Global Value Domains

Now you should have a newly created EDMA project up and running with one global value domain called Name. Now lets define some more global value

domains. We need a Date value domain that contains a year, a month and a day of month. All values from this value domain should be valid dates. First lets create the value domains Year, Month and DayOfMonth:

```
// Year, Month and DayOfMonth
ValueDomain Year : Integer[0..9999]
ValueDomain Month : Integer[1..12]
ValueDomain DayOfMonth : Integer[1..31]
```

Add those value domains to the Common.edma file. You can make comments in edma files just like in java files with // and /* ... */.

Before we continue, we will take a quick look at how these value domains work. Run Make and refresh the project. Now create a new Test class in the default package that looks like this:

```
import tutorial.divingschool.generated.valuedomains.Year;
import edma.valuedomains.exceptions.InvalidValueException;

public class Test
{
  public static void main(String[] args)
  {
    //A valid year
    Year foo = Year.create(1988);
    System.out.println(foo.toString());
    foo = Year.fromString("1807");
    Year bar = Year.create(1807);
    System.out.println("foo equals bar: " + foo.equals(bar));
    System.out.println("Is -11 a valid year: "
      + Year.isValidYear(-11));
    System.out.println("Is 1990 a valid year: "
      + Year.isValidYear(1990));
    //Lets try to make an invalid year:
    try
    {
      bar = Year.create(-54);
    }
    catch(InvalidValueException e)
    {
      System.out.println(e.getMessage());
    }
  }
}
```

As you can see from this code there are a few static methods on the Year class that can be used to create new values from the Year value domain. There is also a method to get a value from a human user through a simple text based terminal:

```
Year.fromTerminal(new SimpleTerminal());
```

You can also take a look at the generated Year.java file to see all the methods.

Now we can create the Date value domain as a *Struct* value domain that uses the Year, Month and DayOfMonth value domains. Add the following to the Common.edma file:

```
//Date
ValueDomain Date : Struct
{
        year : Year,
        month : Month,
        day : DayOfMonth
}
```

Notice that in a *Struct* value domain you can only refer to value domains that are defined elsewhere. The order is not important, you can define Year, Month and DayOfMonth before or after Date or even in a separate file if you want. But you can not use the basic value domains like String, Integer etc in a *Struct* value domain. So this is NOT possible:

```
//Invalid definition of Date
//Will NOT compile
ValueDomain Date : Struct
{
        year : Integer[0..9999],
        month : Integer[1..12],
        day : Integer[1..31]
}
```

Now run Make and refresh the project. The Date value domain is a *Struct* value domain that are build from other value domains. Each *field* in a *Struct* value domain consist of a name and a value domain. A *field* in a struct can be optional, which means that the value may be *null*. To make a field optional all you have to do is add a question mark after the name of the field. Here is an example of a Struct value domain with some optional fields:

```
//Value Domain with optional fields
ValueDomain FooBar : Struct
{
        foo? : SomeValueDomainName,
        bar : SomeValueDomainName,
        foobar? : AnotherValueDomainName
}
```

Here foo and foobar are optional so they may be *null*. If a field is not optional then a NullPointerException is thrown if it is set to *null*.

Struct value domains has a natural order where the significance of the fields follows their position in the *Struct*, with the first field being the most significant and the last field least significant. So if we had put the fields in another order, then dates would not sort properly when using their natural order. Here is some examples of how you can create values in the Date value domain.

```
import tutorial.divingschool.generated.valuedomains.Date;

public class DateTest
{
  public static void main(String[] args)
  {
    //Create a date with the create() method
    Date foo = Date.create().year(2012).month(2).day(15);
    System.out.println(foo.toString());
    //Create a date from a string representation of the date
    Date bar = Date.fromString("(2012, 3, 4)");

    //What happens if we create an invalid date like this 2012-2-30
    Date notValid = Date.create().year(2012).month(2).day(30);
    System.out.println(notValid.toString());
  }
}
```

Notice how the creation uses fluid interfaces to make it easier too read the code. This can be very useful if you create struct value domains with many fields. There is a separate interface for each field you set that returns the interface to set the next field, so it is not possible to leave out a field without

getting a compile error from the java compiler. Try it out, write Date foo = Date.create().year(1988).day(15) and see that the compiler will not accept it because the month is missing. Also the order of the fields is fixed when creating the values, so Date.create().month(5).year(1788).day(7) will not compile either.

At first this strictness in the creation can seem a little rigid, but if you later on add another field to a struct value domain, then the compiler makes sure that you take this new field into account everywhere you create values with that value domain.

If you want, try to create a TreeSet<Date> and put some dates into it, they should be correctly sorted by their natural order when you iterate over the set.

When creating a Date value, it is automatically tested that the year, month and day are within the allowed limits for their value domains, but what about invalid dates as 30/2/2012? These are not caught at the moment. We can add a user implemented constraint to do this:

```
//Date
ValueDomain Date : Struct
{
        year : Year,
        month : Month,
        day : DayOfMonth
} Constraints[validDate]
```

This is done by adding the keyword *Constraints* followed by a comma separated list of names in square brackets. In this case we only added one constraint. Now run Make and refresh the project. Try to run the DateTest class above again and see what happens. It should throw an exception saying that the validDate constraint is not implemented and where you can implement it. Navigate to the class mentioned in the exception. It should look like this:

```
package tutorial.divingschool.usercode.valueconstraints.date;

import tutorial.divingschool.generated.valuedomains.Date;

/**
 * This class is the implementation class for the Date constraint
     validDate
 * No description given
 */
public class ValidDate
{
    /**
     * Checks the validDate constraint for the Date value domain.
     * No description given
     * @param date  The instance value to be checked
     * @return      the reason the constraint is violated, or <tt>null</
        tt> if
     *              the constraint is not violated
     */
    public static String checkValidDate(Date date)
    {
        // Implementation of constraint validDate
        // WARNING : Any code outside the following begin and end tags
        // will be lost when re-generation occurs.

        // EDMA_non-generated_code_begin

        //TODO: Implement the constraint validDate here...
        return "Constraint not implemented. Implement in tutorial.
            divingschool.usercode.valueconstraints.date.ValidDate";
```

```
        // EDMA_non-generated_code_end
    }
}
```

The class is in a sub package of the *usercode* package. All classes that you need to put code into will be placed in the usercode package or its sub packages. Now lets implement the validDate constraint to only allow valid dates. This involves checking for leap years. Be sure to only put your code between the *EDMA_non-generated_code_begin* and the *EDMA_non-generated_code_end* tags, since anything outside these tags will be re-generated when you run Make (except any imports you might add, these will be preserved too). Now try to add the following code between the *EDMA_non-generated_code_begin* and the *EDMA_non-generated_code_end* tags:

```
// EDMA_non-generated_code_begin
int day = date.day().value();
if(day <= 28) return null;
switch(date.month().value())
{
  case 2:
    if(day > 29) return "February can not have more than 29 days";
    // day == 29
    int year = date.year().value();
    if(year % 4 == 0 && (!(year % 100 == 0) || year % 400 == 0)) return null;
    return Integer.toString(year)
      + " is not a leap year, so February can not have 29 days";
  case 4:
  case 6:
  case 9:
  case 11:
    if(day <= 30) return null;
    return "April, June, September and November can not have more than 30 days";
  default:
    return null;
}
// EDMA_non-generated_code_end
```

As you see here the return value is a *String*. You should return *null*, if the value is valid, otherwise you should return an error message that says why the value is not valid. After you have updated this file, try to run the DateTest again. This time only the invalid dates like 30/2/2012 should fail.

With user implemented constraints you can make value domains that are very precise in which values they accept and which they do not accept. You should always try to make your value domains as precise as possible, because then you will catch malformed input as early as possible. The value domains are your guarantee that the input to and output from your *actions* and *views* are valid. As an example of a very specialized value domain, we will make a value domain called FunnyInt that only contains positive integers that are even, but not dividable by 10:

```
//FunnyInt
ValueDomain FunnyInt : Integer[1..MAX] Constraints[even, notDividableBy10]
```

Add the above to your Common.edma file and then run Make and refresh the project. Now find the file ${project}.usercode.valueconstraints.funnyint.Even.java and replace the code between the *EDMA_non-generated_code_begin* and the *EDMA_non-generated_code_end* tags with this:

```
if(funnyInt.value() % 2 == 0) return null;
return funnyInt.value() + "␣is␣not␣even!";
```

In the same sub package find the file NotDividableBy10.java and replace the code between the *EDMA_non-generated_code_begin* and the *EDMA_non-generated_code_end* tags with this:

```
if(funnyInt.value() % 10 == 0) return funnyInt.value() + "␣is␣dividable␣by␣10!";
return null;
```

Make a test class and try out your new FunnyInt value domain.

We actually do not need the FunnyInt value domain in the course registration system after all, so feel free to delete it from the Common.edma file and regenerate. When you do this, you will notice that the Even.java and NotDividableBy10.java files are still there, but now they do not compile because the FunnyInt value domain no longer exists. You will have to delete these files yourself. EDMA will never delete files inside the usercode package or its sub packages. Only stuff outside the *EDMA_non-generated_code_begin* and the *EDMA_non-generated_code_end* tags will be overwritten by EDMA in the *usercode* package and its sub packages. But keep in mind that files you put in the *generated* package or its sub packages will be deleted by EDMA.

Now lets make the value domains that we need to create the Person Kind. The Person Kind should have a name, an email, a mobile phone number and a credit balance. The credit balance should be a non-negative integer, since we do not allow negative credit. We already have a Name value domain that is just a String of length 1 or more. Lets make the other value domains:

```
//Email
ValueDomain Email : String[3..MAX]
//Mobile
ValueDomain Mobile : String[8]
//Credit
ValueDomain Credit : Integer[0..MAX]
```

As seen here the String value domains can both have a length range like 3..MAX and 3..256 or it can just have a single length value like in the mobile value domain. This means that a value from the mobile value domain must be a string that are exactly 8 characters long. We could also have used the notation 8..8 which is the same.

String value domains can also have regular expressions added to them. These are added in square brackets after the length constraints. The syntax for the regular expressions should be just like you would write it in java as a hard coded string surrounded by double quotes. Lets add a regular expression to the Email value domain and the Mobile value domain:

```
//Email
ValueDomain Email : String[3..MAX]
["[\\w-]+(\\.[\\w-]+)*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})"]
//Mobile
ValueDomain Mobile : String[8]["[0-9]+"]
```

This is not a perfect email regular expression, but it serves fine as an example of how to use regular expressions on String value domains. In the mobile regular expression we do not need to test that the length is 8, since this is already being

tested in the length constraint. You should try it out now by making some valid and invalid emails and mobile phone numbers in a test class.

The last thing we will look at before we move on to the data model is how to create a *List* value domain. As an example we will create a value domain where the values are lists of dates:

```
//DateList
ValueDomain DateList : List<Date>
```

Run Make and refresh the project. Now here is some examples of how you can create and use values from the DateList value domain:

```
DateList myDateList = DateList.begin()
                .add(Date.fromString("(2012, 2, 1)"))
                .add(Date.fromString("(2012, 4, 5)"))
                .add(Date.fromString("(2012, 6, 26)"))
                .end();
System.out.println(myDateList);
for(Date date : myDateList)
{
  System.out.println(date);
}

DateListBuilder builder = DateList.begin();
for(int i = 1; i <= 29; ++i)
{
  Date d = Date.create().year(2012).month(2).day(i);
  builder.add(d);
}
DateList datesInFebruary = builder.end();

System.out.println("The dates in february 2012:");
for(Date date : datesInFebruary)
{
  System.out.println(date);
}

DateList someDates = DateList.fromString("((1999,1,3),(1988,2,5),(1745,3,4))");
System.out.println(someDates);
DateList empty = DateList.fromString("()");
System.out.println(empty);
```

It is important to notice that like all other values from value domains, list values are also immutable. The value is not created until the end() method is called on the builder interface and when the value is created, it can never be changed. So in the shown example the DateList datesInFebruary will always contain the same dates in the same order that they where added to the builder.

List value domains has a natural order, where the elements are compared one by one from the beginning of the lists. So if we have lists of integers, then (1,3) is greater than (1,2,3), but (1,2) is smaller than (1,2,3). This is the same way that Strings in java is naturally ordered as lists of characters.

This is all you need to know about value domains for now.

A.3 Defining the Data Model

Create a new file in the edma folder called CourseRegistration.edma. This file will contain our definition of the course registration data model. Write the following into the file:

```
DataModel CourseRegistration
{
}
```

Now run Make and refresh the project. In the package explorer navigate to the package *generated*. Inside this package you will find a package called *courseregistration*. Inside this package you should find the following files:

- *CourseRegistration*.java - This is the external API to instances of the data model. Right now this interface is empty because we have no actions or views in our data model definition yet. Later on when we add actions and views to the data model, they will show up in this interface.
- *CourseRegistrationInstance*.java - This is the interface to a specific instance of the data model. This interface has methods to start and stop the instance and to get the external API from the instance.
- *CourseRegistrationFactory*.java - This is the factory class where we can get access to the instances of our data model. Each instance of a data model is identified by a name. The factory provides methods to create new instances, to test whether an instance with a given name exists and to get access to an existing instance.
- *CourseRegistrationViewer*.java - This is the internal interface that views on the data model has access to. It will get all the methods and functionality that we need to extract information from the data model instance. Right now the data model definition is empty, so this interface is also empty.
- *CourseRegistrationUpdater*.java - This is the internal interface that actions on the data model has access to. It extends the *CourseRegistrationViewer* interface, so actions can also extract information from the data model, but then it adds methods and functionality to update the state of the data model instance. Right now this interface is empty because the data model definition is empty.

Inside the *courseregistration* package you should also find two sub packages called *remote* and *test*. The *remote* package contains classes that makes it possible to set up a server that handles access to a data model instance, and a proxy class that clients then can use to access the data model instance on the server. The *test* package contains a class that takes an API interface to a data model instance and then creates an interactive console where *actions* and *views* on the data model can be called.

Before we can try out all these things, we need to have something in our data model definition. To start out with we will create a very simple data model with a Person Kind, an action that can create new persons and a view that can get a list of all the persons.

A.4 Kinds

First we define the Person Kind. Update your CourseRegistration.edma file to look like this:

```
DataModel CourseRegistration
{
        Kind Person
        {
                name : Name,
                email : Email,
                mobile : Mobile
        }
}
```

Again run Make and refresh the project. Now take a look inside the *...generated.courseregistration* sub package again. You should now see two new sub packages that have been created: *kinds* and *valuedomains*. If you take a look into the *valuedomains* sub package you should find the files: *PersonID*.java, *Person*.java and *PersonList*.java. These are value domains that have been automatically created from the Person Kind. The *PersonID* value domain contains a long value that are used to identify a person entity inside the data model instance. All Kinds have an implicit id value beside the values that are defined explicitly. The *Person* value domain is a struct value domain that contains this id value plus all the values that are defined explicitly in the Person Kind. The *PersonList* value domain is a *list* value domain where the elements are values from the *Person* value domain. All these auto generated value domains are local to the data model which means that you can not use them in global value domains or value domains that are local to other data models. You can also make your own local value domains by defining them inside the data model's { and }. If you want the *PersonID*, *Person* and *PersonList* to be global value domains, so you can use them in other global value domains or in other data models you can write Publish after the kind name like this:

```
DataModel CourseRegistration
{
        Kind Person Publish
        {
                name : Name,
                email : Email,
                mobile : Mobile
        }
}
```

You can also make them global with another name to avoid name clashes with other global value domains. This is done with the As keyword like this:

```
DataModel CourseRegistration
{
        Kind Person Publish As MyPerson
        {
                name : Name,
                email : Email,
                mobile : Mobile
        }
}
```

Now the generated value domains will be global with the names: *MyPersonID*, *MyPerson* and *MyPersonList*.

Before we go on you will need to know a few things about what a data model instance is. A data model instance should be viewed as a sealed container that contains some data that might change over time. It is not possible to see or update the data from outside the container. But the container provides *actions* and *views* that can go inside the container and look at the data and the *actions* can also make changes to the data. But they can not take any references to the data outside the container. So the only way a *view* or an *action* can get information out of the container is by taking snapshots of the data and then bring these snapshots to the outside. A snapshot can be thought of as an immutable picture of how the data looked when it was taken. In EDMA we use values from value domains to capture these snapshots, since these values are immutable. So the generated PersonID, Person and PersonList value domains can be used to bring information from inside the data model instance to the outside.

The *attributes* in a kind can also be made optional with a question mark, just like the fields in a *Struct* value domain.

Now lets look into the *kinds* sub package, there should be a sub package called *person*. Inside the *person* sub package you will find the internal interfaces that can be used by the *views* and the *actions* when they operate inside the data model instance. The sub package should contain these files:

- PersonViewer.java - This is the view interface to a person entity inside the data model instance. This interface contains methods to get snapshots of all the attributes in the entity. If you look into this interface you will find the methods: getID(). getName(), getEmail() and getMobile(). The values returned by these methods are snapshots of the entity attributes. There is also a method named snapshot() that will take a snapshot of the entire entity as a value from the Person value domain described earlier.

- PersonUpdater.java - This is the update interface to a person entity inside the data model instance. Right now this interface is empty because we have not allowed any of the attributes in the Person Kind to be updated. Lets try to make the *mobile* attribute updateable. This is done by adding a + after the name of the attribute like this:

```
DataModel CourseRegistration
{
        Kind Person
        {
                name : Name,
                email : Email,
                mobile+ : Mobile
        }
}
```

  Now run Make again and refresh. The PersonUpdater interface should now contains a method called beginUpdate() that returns another interface where the updateable attributes can be set and a save() method to end the updates to the attributes. Right now it might seem a little overkill that you need to call beginUpdate().setMobile(...).save() to update the

mobile phone number. But later on when we introduce the Unique index it will be more clear why it is done in this way.

- PersonSet.java - This interface represents a set of person entities (each of them represented by the PersonViewer interface). You can iterate over the set, you can order the set by any of the attributes and you can suborder by any attribute. You can also perform set operations like union, intersection and subtraction. Sets are immutable, so every time you change something, you actually get a new set. The set also has a snapshot() methods that returns it as a value from the PersonList value domain. The interface is made so the runtime system can choose to delay all set operations like union, intersection, subtraction, ordering etc until the set is actual iterated over or accessed in other ways. This makes it possible for the runtime system to optimize the set operations.

- PersonFilter.java - This is a filter interface that can be used to implement user defined filters to perform specialized searches in sets. The interface has a single method called *accept* that takes a PersonViewer interface as parameter and returns a boolean.

- PersonKind.java - This is the interface to the collection of all person entities in the data model instance. It has methods to get a specific entity by its ID and to get all the entities as a set.

A.5 Views and Actions

The entities in the data model instance can only be accessed through *views* or *actions*. *Views* can only extract data from a data model instance while *actions* can both extract data and make changes to the data inside a data model instance. We will start by making an action that can create a new person kind.

Create a new file in the *edma* folder named CourseRegistrationAPI.edma that contains this:

```
DataModel CourseRegistration
{
        Action createPerson
        {
                Description:
                        "Creates a new person"
                Input:
                        name : Name,
                        email : Email,
                        mobile : Mobile
                Output:
                        id : PersonID
        }
}
```

This defines an action that can create a new person. It has a description, some input parameters and an output parameter.

Now run Make and refresh the project.

First we have to implement the action to do what we want, namely create a new Person entity. Locate the file: tutorial.divingschool.usercode.models.

courseregistration.actions.CreatePersonUserImpl.java. It will intentionally have a compile error, so it should be easy to locate it.

```
.
.
.
    /**
     * Execution of the action
     * @param upd  Update interface
     * @return     Return 0 to commit or one of the error codes to roll back
     */
    public int execute(CourseRegistrationUpdater upd)
    {
        // Implementation of createPerson
        // Return one of the following error codes:
        // OK

        // If an error needs extra explanation, use: setErrorDescription("Extra info");

        // WARNING : Any code outside the following begin and end tags
        // will be lost when re-generation occurs.

        // EDMA_non-generated_code_begin

        "TODO : put your implementation of createPerson here...";

        // EDMA_non-generated_code_end
    }
.
.
.
```

The implementation should go between the *EDMA_non-generated_code_begin* and *EDMA_non-generated_code_end* tags in the *execute* method. The execute methods has a single parameter which is an interface to the data model instance. Since this is an action, the interface is an update-interface that can change the state of the data model. The execute method returns an int, which is a status code. Since we have not yet defined any error conditions, we should simply return OK (which is always defined as 0). Replace the implementation of the execute method with this:

```
// EDMA_non-generated_code_begin
PersonUpdater person = upd.newPerson()
                .name(in_name)
                .email(in_email)
                .mobile(in_mobile);
out_id = person.getID();
return OK;
// EDMA_non-generated_code_end
```

As you can see the class has defined all the input parameters prepended with "in_". These parameters are automatically initialized by the EDMA system before the execute method is called. The output parameters are also defined prepended with "out_". The values of the output parameters must be set in the execute method by you (unless you return a non-zero error-code, then you can ignore the output parameters, but more on that later).

Before we test the create person action, lets make a view that returns all persons, so we can see if they actually gets created. In CourseRegistrationAPI.edma add this view to the data model CourseRegistration:

```
        View getAllPersons
```

```
{
        Description:
                "Returns a list of all persons"
        Output:
                personList : PersonList
}
```

This view takes no input and returns a list of all persons. Run Make and refresh, then locate the file: tutorial.divingschool.usercode.models.courseregistration.views.GetAllPersonsUser and put the following implementation into the *execute* method:

```
// EDMA_non-generated_code_begin
out_personList = view.getPersonKind().getAll().snapshot();
return OK;
// EDMA_non-generated_code_end
```

Notice that the parameter to the execute method is a *view* interface because this is a view and not an action, so it is not allowed to change the state of the data model instance.

A.6 Try it out

We will now go through all the steps needed to create an instance of the CourseRegistration data model and manipulate it through the actions and views we have created. To do this create a class called TryIt in the default package with a main method.

The first thing we do is to create a runtime factory. The runtime factory provides the runtime execution logic to each data model instance. There can be many different implementations of the runtime factory with different strategies for execution and persistence. The example runtime factory provided with this version of EDMA features a thread-safe pipelined execution and single file persistence. The argument to the constructor is the directory where the instances should be persisted. The runtime factory is generic code that is not generated from any specific data model.

```
RuntimeFactory rtfactory = new RuntimeFactory("C:/tmp");
```

Now that we have a runtime factory, we need to create our environment and connect it to the runtime factory:

```
DivingSchool ds = new DivingSchool(rtfactory);
```

The environment is an auto-generated class that provides access to instance factories for each data model defined in the environment. At the moment we only have one data model in our environment, the CourseRegistration data model. A data model can have several independent instances, so each data model has an instance factory that can create and delete instances of that particular data model.

Now lets get the instance factory for our CourseRegistration data model:

```
CourseRegistrationFactory crf = ds.getCourseRegistrationFactory();
```

The instance factory controls the instances of the data model. Each instance is identified by a name, for now we just need one instance that we will call

"MyInstance". If the instance already exists, we will reuse it, if it does not exist, we will create it:

```
CourseRegistrationInstance instance = null;
String instanceName = "MyInstance";
if(crf.exists(instanceName))
{
  instance = crf.getInstance(instanceName);
}
else
{
  instance = crf.newInstance(instanceName);
}
```

NOTE: An instance of a data model contains data that follow the structure of that particular data model. This means that if you change the data model, then the instances are no longer usable. This means that every time you make changes to the structure of your data model you should also delete any instances you have stored. You may create, delete and modify the actions and views on the data model and still reuse the instances, just be careful not to change any value domains that are used in the kinds.

From the instance we can get the API that lets us execute actions and views on the instance. With the example runtime factory that we are using, instances are thread-safe, so the API can be operated from several threads simultaneously. Now lets get the API from our instance:

```
CourseRegistration cr = instance.getAPI();
```

Before we can call any action or views on the instance, it needs to be started. Instances can be started and stopped. When an instance is started, its internal worker-threads are created and ready to execute actions and views. When we are finished using an instance it should be stopped.

Now lets fire up the instance:

```
instance.start();
```

That's it! Now we can start executing actions and views on our data model instance. Lets try out the action "CreatePerson":

```
cr.createPerson(Name.create("John Doe"),
        Email.create("johndoe@foo.bar"),
        Mobile.create("12345678"));
```

In order to check that we actually did create a new entity, lets try the view "GetAllPersons":

```
PersonList list = cr.getAllPersons().getPersonList();
System.out.println(list);
```

Since actions and views can have multiple return values and also returns a status, each action or view has its own result interface on which the status and the return values can be accessed. So the getAllPersons() methods returns a result on which we then can access the return value "PersonList" by calling getPersonList().

All value classes has a default toString() method so we can just print out the entire list with System.out.println(list).

Now we just need to stop the instance to terminate the program:

```
instance.stop();
```

First time this is run it will create a new data model instance with the name "MyInstance" and add a person to that instance. If it is run again it will just load the existing instance and add another person to that. Try it out...

If you look in the directory you gave to the runtime factory (in this example it was "C:/tmp"), you should see a file called "myinstance.data". This file contains the persisted data for the instance.

Another way to test the data model is to use the auto-generated terminal test program. The terminal test program takes an interface to the API and an interface to a terminal (a terminal here just means a simple text-in, text-out user interface). It lets the user call the actions and views from the terminal. To use the auto-generated terminal test program, replace the lines between *instance.start()* and *instance.stop()* with this line:

```
new CourseRegistrationTest(cr, new SimpleTerminal()).start();
```

Then run the program. Now you can create new persons and see the list off all persons through the terminal test program.

From now on, when you create new actions or views and run Make, the terminal test program will be regenerated to support the new methods.

A.7 Indexes

Before we continue, you should delete the instance persistence file "myinstance.data". This is important because we are going to make changes to the data model.

We will now add some indexes to our data model. There are 3 types of indexes in EDMA: unique index, equal index and compare index. The unique index is used to force certain fields or combinations of fields to be unique within a kind. For example we could make the email and mobile fields unique in the person kind. To do this change the definition of the person kind to look like this:

```
Kind Person
{
        name : Name,
        email : Email,
        mobile : Mobile,
        Unique(email),
        Unique(mobile)
}
```

This means that now 2 different persons can not have the same email or the same mobile number. If you run make and refresh the project, you will see some changes in the generated files:

1. In CourseRegistrationUpdater.java, the newPerson(...) method now declares to throw an UniqueException. EDMA will throw this exception if you try to create a person with an email or mobile number that already exists.

2. The PersonKind interface has got two new methods: getFromEmail(...) and getFromMobile(...).

Since it is generally considered too be bug if actions or views throws exceptions, EDMA provides a better way of handling this. In the definition of the "CreatePerson" action we can declare that the action may fail on certain input. We do this by defining two error code:

```
Action createPerson
{
        Description:
                "Creates a new person"
        Input:
                name : Name,
                email : Email,
                mobile : Mobile
        Output:
                id : PersonID
        ErrorCodes:
                1 - "Email already exists",
                2 - "Mobile already exists"
}
```

Now run make and refresh the project again. The CreatePersonUserImpl class now has two extra "static final int" that represents the added error codes. You can now alter the implementation of the action "CreatePerson" to look like this:

```
EDMA_non-generated_code_begin
//Check that email is unique
if(upd.getPersonKind().getFromEmail(in_email) != null)
{
  return EMAIL_ALREADY_EXISTS;
}
//Check that mobile is unique
if(upd.getPersonKind().getFromMobile(in_mobile) != null)
{
  return MOBILE_ALREADY_EXISTS;
}

PersonUpdater person = upd.newPerson()
                .name(in_name)
                .email(in_email)
                .mobile(in_mobile);
out_id = person.getID();
return OK;
// EDMA_non-generated_code_end
```

Now the action first checks that the email and mobile number are indeed unique, before it tries to create the new person. The isolation property of the ACID guarantees will make sure that no other actions can race in and create a person in between we checked the uniqueness and we created the new person.

Only the Unique index alters what data we can put into the data model instance. The equal and compare indexes only helps making it easier and faster to access certain sets of entities. If you want to make it easy and fast to get the set of all entities in a kind where one or more attributes are equal to certain values, then you should use the equal index. If you also want to make it easy and fast to get entities where certain attributes are less than, greater than or in between certain values, you should use a compare index.

Equal and compare indexes are defined with the "Equal" and "Compare" keyword just like unique indexes are defined with the "Unique" keyword. You

should try them out and take a look at the methods they add to the kind interface.

Indexes can be defined on a single attribute or on multiple attributes separated by commas.

A.8 Relationships

You can also define relationships between kinds. There are 3 main types of relationships: "one to one", "many to one" and "many to many" ("one-to-many" is just a "many-to-one" where the kinds are reversed). Relationships are defined using the Relation keyword in the EDMA definition language. A relationship keeps track of individual connections between entities from the two kinds that participate in the relationship. An example could be a relationship between a student and a course. We could call this relationship StudentEnrollment and every connection in the relationship would represent that a specific student entity is enrolled on a specific course entity.

The relationship definition would look like this:

```
Relation StudentEnrollment Course >-< Student
```

And it would lead to the creation of the method:

```
public CourseSet getCourseSet();
```

in the StudentViewer interface. The methods returns the set of courses that the student are currently enrolled on. In the CourseViewer interface we would get this method:

```
public StudentSet getStudentSet();
```

that would return the set of students that are currently enrolled on the course. In the CourseUpdate interface we would get these methods:

```
public boolean addStudent(StudentViewer student);
public boolean removeStudent(StudentViewer student);
```

The StudentUpdate interface does not get methods to add or remove courses. The add/remove methods are always added to the kind that appears first in the definition.

If each course could have one teacher that would be a many to one relation and the definition could look like this:

```
Relation StudentEnrollment Course >-- Teacher
```

Try for yourself to create a "many-to-one" and see what methods that are created on the interfaces of the involved kinds.

But what if we did not have a student kind and a teacher kind, but just a person kind?

In this case we would need to distinguish the role of the person in the two relationships:

```
Relation StudentEnrollment Course >-< Person : student
Relation StudentEnrollment Course >-- Person : teacher
```

The name after the colon is the role name. Try it out and see how it alters the names of the created methods on the involved kinds. If no role name is provided, the role is asserted to be the same as the name of the kind. So with the example of `Course >-< Person`, if no role name is provided for the Person, it just has the role of a person, which is very general.

A "one-to-one" relationship is defined by **''---''** as you might have guessed. Try it out and see which methods you will get.

Note that all the generated interfaces have meaningful javadoc (or at least they should have, lets us know if some is missing). So you can always experiment with different data model definitions and see what gets generated and what it does. Hopefully it will make good sense in the context of the data model.

A.9 Remote Access

The generator also generates java code that let you use the data model instance from a different machine through a socket connection. The code consists of:

- A simple server that uses a thread pool to handle incoming requests and relate them to the data model interface
- A proxy that implements the data model interface and relates all method calls to the server

The follow code shows how you create a server from a data model instance interface:

```
//Make sure the instance is running...
CourseRegistration intf = instance.getAPI();
int port = 1234;
Server server = new Server(port, new CourseRegistrationServerInstance(intf));
server.start();
```

And this shows how you would access the interface on the client side:

```
CourseRegistration intf = new CourseRegClientInstance("localhost", 1234);
```

(Replace "localhost" with the IP of the server). You can now use the interface on the client side, just as if the data model instance was running locally.