jDomainApp:
A Java Domain-Driven Software Development Framework

Technical report v5.2
18/06/2019

Duc Minh Le
duclm@hanu.edu.vn
Faculty of IT, Hanoi University

Abstract. In two recent work, we proposed a novel tree-based, domain-driven software architecture for interactive domain-driven software based on the Model-View-Controller (MVC) design architecture, and a domain-driven domain class specification language for capturing the essential data requirements of a domain in its domain model. In this paper, we discuss in detail a Java-based software development framework, named jDomainApp, that realise these work. We also briefly describe a light-weight software development method that uses the framework to automatically generate software. The framework produces a software architecture consisting of a set of MVC modules, each of which is represented as a Java class that is structured from three component Java classes: `Controller` (the controller), `View` (the view), and a domain class (the model). The modules are realised in the framework using Java generic and annotation. The domain classes are mapped automatically to relational tables that are used for storing the domain objects.

1. Introduction

Software that are designed using the Model-View-Controller (MVC) architecture [1], [2] consist of three components: model, view, and controller. The model component encapsulates the data about the entities of interest that a software needs to capture, store and process. The view component is responsible for presenting the data to the user on a user interface and to make it easy for her to interact with it. The controller component is responsible for handling the user interaction on the view and for updating the model if required. The model and the view also work with each other to present to the user any changes to the model data.

In a recent paper [3], we proposed a novel tree-based, domain-driven software architecture for interactive domain-driven software based on the Model-View-Controller (MVC) design architecture. A key feature of this architecture is the representation of a software as a set of modules, each of which is a class that is structured from three classes: `Controller` (the controller), `View` (the view), and a domain class (the model). The modules are combined using containment tree to form composite modules that fulfil complex software requirements.

In another recent work [4], we proposed a domain class specification language for capturing the essential data requirements of the domain in a set of Java domain classes.

In this paper, we discuss in detail a Java-based software development framework, named JDOMAINAPP, that combines the above work and realise them in the Java language. This results in a software architecture consisting of a set of MVC modules, each of which is represented as a Java class that is structured from three component Java classes. We present a *reduced* design of the module class using a combination of `Controller` and a module configuration class. The modules are combined using containment tree to form composite modules that fulfil complex software requirements. The framework uses two domain-driven design features of the Java language, namely generic and annotation, to realise the modules and their containment trees. The domain classes are mapped automatically to relational database tables that are used for storing the domain objects. We describe and demonstrate a light-weight software development method that uses the framework to significantly reduce the development effort by enabling the automatic generation of a software.

The paper is structured as follows. Section 2 describes a software example that will be used throughout this paper. Section 3 gives an overview of the design features of the core framework classes. Sections 4-6 explain in detail the design of these classes. Sections 7 and 8 present the design of a configuration method for the software that are developed in the framework. Section 9 describes the light-weight development method. Section 10 explains the related work and, finally, Section 11 concludes the paper.

2. Software Example: COURSEMAN

2.1. Functional requirements

COURSEMAN is a simplified course management software. The primary purpose of this program is to support the **basic activities**, such as to enrol a student into a pre-defined course module each semester, and to allow the administrative staff of the faculty to enter the module marks for each enrolment and to compute the final grade from these marks. In addition to the basic activities, the program needs to support **composite activities**, which are activities that involve the performance of a number of basic activities in a pre-defined order. The first such activity is **student registration**, which involves first capturing a student's profile details and then enrolling this student into a course module. Another composite activity that the program needs to perform is **class management**, which is responsible for arranging enrolled students into one or more classes. The third composite activity is **enrolment management**, which manages the entire process constituting of student registration and class management.

As far as reporting is concerned, the program needs to generate a **student profile report**, which includes details about id, name, dob, and email of the students whose names match a user-specified name.

2.2. Data requirements

Students have the following attributes: id, name, dob (the abbreviation for date of birth), address, email. Attribute id is a unique student identifier which is generated automatically by the system using

the formula: the letter "S" followed by a number, which is auto-incremented from the current year. For example, the first student to be registered into the program in the year 2010 will have the id of S2010. The second student of that year has the id of 2011, and so on. Students are grouped into classes.

A **student class** is described by an id (integral identifier) and name.

A **course module** represents a taught course module. It is characterised by the following attributes: code, name, semester (the integral number of a course semester), and credits (the number of credits). The module code is a unique module identifier which is generated by the system using the formula: the letter "M" followed by the next auto-incremented integer starting from a base, which is calculated by multiplying the semester attribute value by one hundred. For example, the first module with the semester attribute value of 1 will have the code of M101, the second module will have the code of M102 and so on. Similarly, the first module with the semester attribute of 2 will have the code of M201, etc.

There are two types of course modules: **compulsory** and **elective**. In addition to the common attributes, an elective module is characterised by another attribute named deptName, which captures the name of the department in which the module is taught. This may be a different department from the home faculty of the students.

An **enrolment** records a fact that a student has registered interest to study a specific module in a given semester (assuming that these modules have already been created). It also holds data about the internal mark, examination mark, and the final grade that the student has obtained in the module Thus, enrolments will have the following attributes: student (the student instance who initiated the enrolment), module (the module instance), internal mark (floating), examination mark (floating), and final grade. Final grade is a single character which must be one of the followings: "E" (excellent), "G" (good), "P" (pass), and "F" (failed).

3. Design Overview

3.1. Class `ApplicationModule`

This class represents the configuration of a software module of a software. An object of this class contains all the information needed to create the M-V-C objects that form a module. In particular, it specifies the `Controller` class that is used to create the `Controller` object of a module and the domain class of the module. In Section 7, we will explain this class in more detail.

As will be explained in the subsequent subsections, our framework uses a *reduced* module class design that uses a combination of `ApplicationModule` and `Controller`.

3.2. Class `Controller`

For historical reasons, this class is implemented as a sub-type of another class called

ControllerBasic. Class ControllerBasic implements all the core functionalities needed by any controller. Class Controller has all these functionalities (through inheritance) and a number of other (enhanced) functionalities. In this note, we will focus on just the core functionalities and so class ControllerBasic will be used in the design diagrams. In the text, however, we will use class Controller, as it fits more nicely with the "controller" component of the framework's architecture.

An object of class Controller is created via the factory method Controller.createController, which is specified with the same parameter list as that of the class's constructor. The first parameter of the constructor is an object of the class named DomainSchema, which provides an interface for manipulating the domain classes of the software. We will explain this class later in Section 4. The second parameter of the constructor refers to the ApplicationModule object.

For reasons that will be explained shortly, Controller is used to serve as the module class. This means that the Controller object of a module will be responsible for creating the View object of that module. The configuration of this View object is specified by the third parameter, which is of type RegionGui. We will explain the view design concepts in Section 5.

The fourth parameter specifies the parent Controller, which is of the main module of the software. This is due to the GUI containment design that will be explained in Section 6.

The fifth and also the last parameter is a Configuration object, which specifies the software-wide configuration settings. Section 7 will explain this configuration in more details.

The two attributes Controller.schema and Controller.module are initialised to the first two arguments of the constructor. Attribute Controller.cls, which is typed Class<C>, is initialised to the domain class that is bound to the Controller.

3.3. Class View

A View is consisted of a number of components, each of which is created from a Region object that is the specified by the RegionGui. The association labelled refers shown in Figure 1 is realised by two attributes: Controller.gui and View.controller.

A View object is created by invoking its 3-parameter constructor shown in the figure. The first parameter specifies the associated Controller object. The second parameter specifies the view configuration object associated to the View. It is mapped to the third parameter of the Controller's constructor, explained in Subsection 3.2. The third parameter specifies the parent View, which contains this View. View containment will be explained in Section 5.

Note that the `View` object created by the constructor is only an empty container. The actual GUI components will not be created until the operation `View.createGUI` is invoked. This invocation is performed from within the operation `Controller.createGUI`.

3.4. Essential Operations

Figure 13 in Appendix 1 shows a number of essential operations of the two core classes `Controller` and `View`.

First, the operation `Controller.run` is used to run a `Controller`, which typically involves an invocation of the operation `Controller.showGUI`. This operation involves making the associated `View` object visible (which consists of an invocation of the operation `setVisible` on the `View` object with the argument `true`.)

In contrast to the operation `showGUI`, operation `Controller.hideGUI` (which invokes the operation `setVisible` on the associated `View` object with the argument `false`) is used to hide the `View` object.

Last but not least, operation `Controller.shutDown` clears the resources being used by a `Controller`. It invokes operation `shutDown` on the associated `View` object and ends the software, if this object represents the main window.

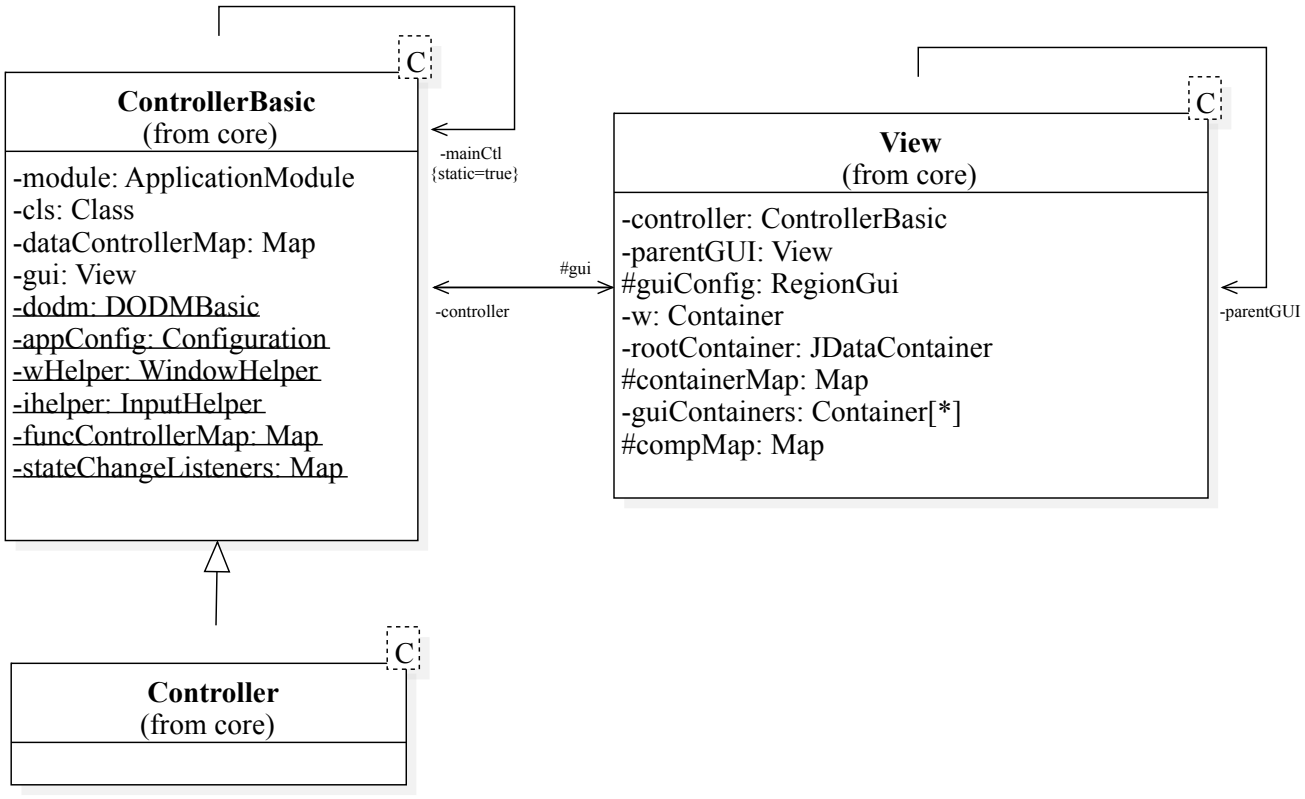

Figure 1: A reduced design class diagram of two core classes: `Controller` and `View`

3.5. Using `Controller` to Represent Module

In designing the framework, we reduce the module class (discussed in [3]) to its most basic capability – one that considers the behaviour of a module object as being *fully* described by the interaction of the three objects that it contains. This leads us to a simplified physical design of the architecture in which the `Module` class is omitted and class `Controller` is used to act on its behalf. The main reason is that the `Controller` object of a module is already the central point of the interaction; and thus promoting it to act on behalf the module is a logical extension of this capability. In particular, this involves assigning to the `Controller` class the additional responsibilities of creating its own object (with a binding to a domain class) and creating a `View` object. These responsibilities were described previously in the Subsections 3.2 and 3.3.

4. Model

In this section, we briefly review the domain-driven design specification (proposed in [4], [5]) for the domain class of the architecture. An in-depth discussion of this topic is presented in [6].

4.1. Design overview

A **domain class** is a class that is designed with domain-specific information. According to [7], existing object oriented language platforms (e.g. NET and Java) provide support for the specification of such information as part of the class design. The idea is to model domain-specific information as a set of meta-attributes, which can be attached to a class, to its members or to both. A **meta-attribute**[1] is a set

1 the term 'meta-attribute' that we use here is synonymous to 'attribute' in [1].

of properties, whose values are specified when the attribute is attached.

In DomainAppTool, five basic meta-attributes are used to model a domain class: `DClass`, `DAttr`, `DAssoc`, `DOpt`. and `AttrRef`. The first meta-attribute is attached to the class, while the others are attahced to class members.

We adapted a design approach in [7], which models a meta-attribute as a class. This class, which we call the **meta-attribute class**, is modelled with the stereotype `<<meta-attribute>>`[1]. The class name is decorated with the *at* ('@') sign. The attachment of this class to a domain class or to one of its members is modelled as an association. To help differentiate this association from the normal class association, we draw it in the class diagram using grey-coloured line that has a filled circle at the opposite end of the meta-attribute class. The association line to a class is connected to one of the sides of the class box. On the other hand, the association line to a class member crosses the class boundary into the class box to connect to the left-most position of the member. The examples given shortly below will illustrate these.

We assume that every property of a meta-attribute is defined with a default value. When it is necessary to save space, only the properties whose values differ from the default are shown in the design model.

4.2. Meta-attribute `DClass`

This meta-attribute has four basic properties: `schema`, `serialisable`, `singleton`, and `mutable`. Other properties can be added if necessary. Figure 2 shows how this meta-attribute is modelled in the class diagram of the CourseMan software. The default values of the four properties are listed in the specification of the meta-attribute for the class `SClass`.

Property **`schema`** specifies the name of the storage schema, in which the domain objects (of the domain class) are stored.

Property **`serialisable`** (which is of type `boolean`) specifies whether or not the domain objects can be serialised to the software's external storage (e.g. a file or database). When this property is `true`, the domain objects can be stored at the storage space, whose name is specified by the property `schema`. In Figure 2, for example, the default values of these two properties mean that all domain classes of the CourseMan software are serialisable to the schema named `App`.

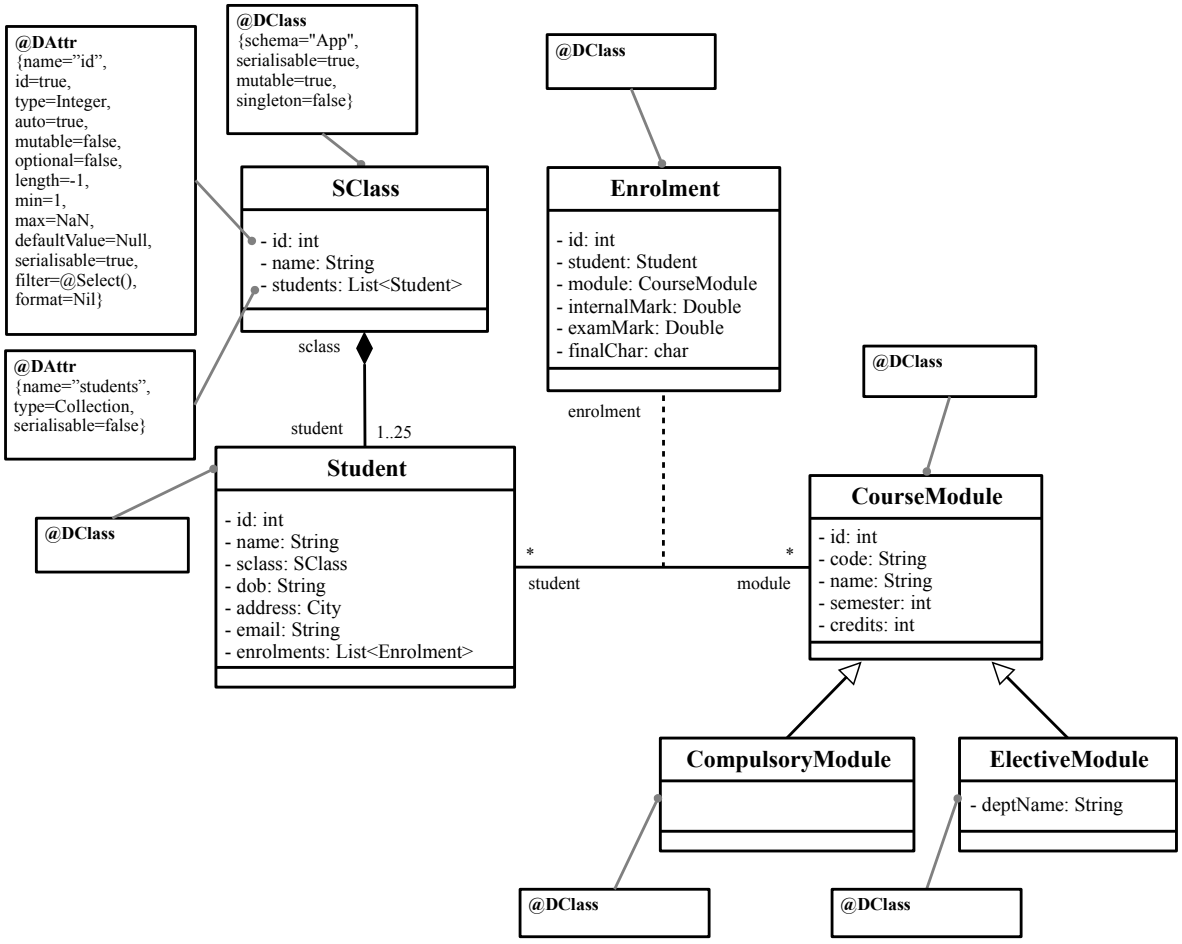

Figure 2: The basic design of the domain classes of the COURSEMAN software.

Property **`mutable`** specifies whether or not the domain objects are mutable (i.e. their states can be changed). The value of this property affects how the domain class and its objects are processed by the software. For example, if the property is `true` then the view of the domain class will present the domain objects as read only, so that the user is not allowed to edit their states. Note that this property applies to all objects of the class, which should be distinguished from another property with the same name of the meta-attribute `DAttr` (introduced shortly).

Property **`singleton`** specifies whether or not there is only one domain object of the domain class that is in use throughout the entire software's run-time. If set to `true`, the domain class and/or its objects can be handled more efficiently by the `Controller`. For example, in our implementation of the framework, the `Controller` reads the domain object of a singleton class from the underlying storage once and keeps it in memory until the software is terminated.

The default values of the two properties `mutable` and `singleton` in Figure 2 mean that all domain classes of the COURSEMAN software are mutable and can have any number of domain objects.

4.3. Meta-attribute `DAttr`

Throughout this paper, we will use the term **domain attribute** to refer to the attributes of a domain class that are specified with a `DAttr` meta-attribute.

This meta-attribute has two sets of properties. The first set specifies the domain constraint

information of an attribute . This set has nine properties: **`name`**, **`type`**, **`mutable`**, **`optional`**, **`length`**, **`min`**, **`max`**, **`defaultValue`** and **`format`**. These properties are rather self-explanatory, though detailed information can be found in [8][2]. The second set has four basic properties: `id`, `auto`, `serialisable`, and `filter`.

Property **`id`** specifies whether or not the domain attribute is an identifier. Note that this is a domain-specific identifier, not the platform-specific object identifier that comes with every class. Property `id` is used to instruct the software on how to extract the domain identifier values of the domain objects. Such values are needed for such tasks as answering a user-specified search query for objects.

Property **`auto`** specifies whether or not the value of the domain attribute is automatically computed by the software. If `false` then the domain attribute's values must be specified by the user.

Property **`serialisable`** specifies whether or not the value of the domain attribute is saved when the domain objects are serialised. This property is used together with the property `DClass.serialisable`.

Property **`filter`** is specifically used for the collection-typed domain attributes that realise the participation of their owner class in 1:M associations with other classes. We will explain association in Section 4.4 and collection-typed attributes shortly below.

Let illustrate the afore-mentioned properties using two domain attributes `SClass.id` and `SClass.students` in Figure 2. Listing 1 below shows the corresponding Java code of the class `SClass` concerning these two attributes. The complete listing of the Java codes of all the domain classes in Figure 2 are given in [6].

The specification for the attribute `SClass.id` includes the values of all the properties. It is also the standard specification for identifier-typed and auto-generated domain attributes. The values of the first five properties are specific to the attribute, those of the remaining properties are the defaults. The value of property `name` is always the same as the attribute name, which is `"id"` in this case. The value of property `id` in this case is `true` because `SClass.id` is the identifier attribute. The value of property `type` is a type constant that best matches the attribute type, which in this case is `Integer` (because `SClass.id` is typed `int`). The value of property `auto` is `true` because, in this case, the values of `SClass.id` are automatically generated by the software. Because of the `auto`'s specification, the value of the property `mutable` must be set to `false` in this case. The value of property `optional` is `false`, which is also the default, means that attribute `SClass.id` must be initialised with a value when an `SClass` object is created.

The specification for the attribute `SClass.students` is the standard specification for collection-typed attributes that realise the participation of its owner domain class in a 1:M association with another class. In this example, the owning domain class of the attribute is `SClass` and the associated class is

2 In addition, FIT students should refer to the relevant PPL lectures on this

Student. The value of property type is the type constant named Collection, because the attribute's type is as such. The value of property serialisable is false, because the collection value of the attribute refers to objects of another class, whose serialisability are determined by the corresponding serialisable specification of that class and not on those of the attribute in question.

```
import java.util.ArrayList;
import java.util.List;

import domainapp.model.meta.DAssoc;
import domainapp.model.meta.DAssoc.Associate;
import domainapp.model.meta.DAssoc.AssocEndType;
import domainapp.model.meta.DAssoc.AssocType;
import domainapp.model.meta.DClass;
import domainapp.model.meta.DAttr;
import domainapp.model.meta.DAttr.Type;
import domainapp.model.meta.DOpt;
import domainapp.model.meta.Select;

@DClass(schema="courseman")
public class SClass {

  @DAttr(name="id",id=true,auto=true,type=Type.Integer,length=6,
     mutable=false)
  private int id;
  private static int idCounter;

  @DAttr(name="name",length=20,type=Type.String)
  private String name;

  @DAttr(name="students",type=Type.Collection,
      serialisable=false, optional=false,
      filter=@Select(clazz=Student.class))
  @DAssoc(ascName="class-has-student",role="class",
      ascType=AssocType.One2Many,endType=AssocEndType.One,
      associate=@Associate(type=Student.class,cardMin=1,cardMax=25))
  private List<Student> students;
}
```

Listing 1: Java specification of the domain attributes of SClass

4.4. Meta-attribute DAssoc

This meta-attribute is used to specify an association end of a binary association between two domain classes. Conceptually, an association is a named set of DAssocs, each specifying one end of it. An DAssoc is attached to the domain attribute that realises its association's end. This attribute is called the **link attribute**.

DAssoc is designed with the following properties: ascName, role, ascType, endType, associate, dependsOn. Figure 3 below illustrates how DAssoc is used to define three associations of the

CourseMan class diagram, which contains an additional class named `City`. This class has an one-one association with `Student`.

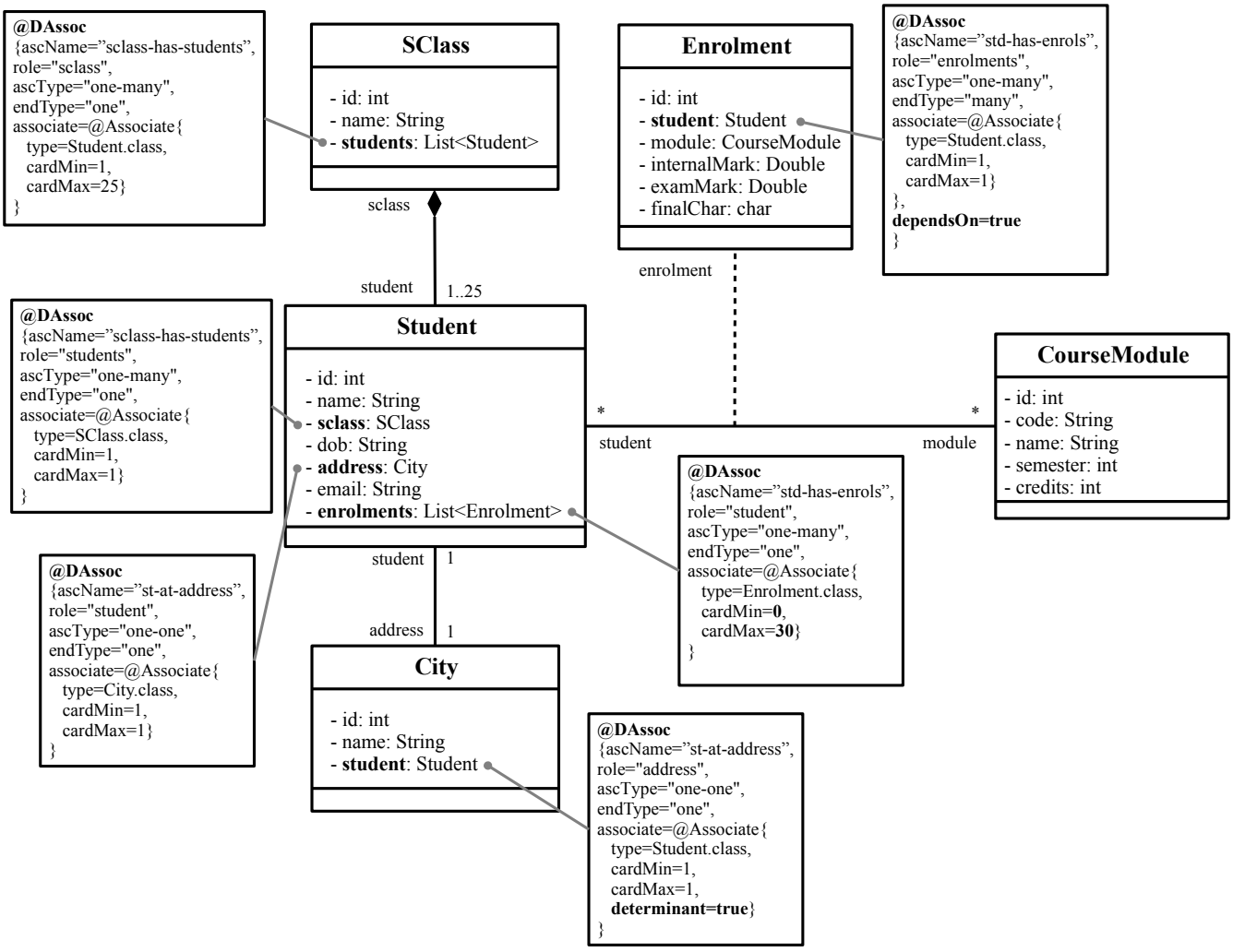

Figure 3: Applying meta-attribute `DAssoc` to three associations of `CouseMan`

Property **`ascName`** specifies the name of the association. It uniquely identifies an association and is therefore used to determine the set of `DAssocs` of that association. A `DAssoc` belongs to this set by way of having the value of this property to be the same as those of other `DAssocs` already in the set. In Figure 3, the value of this property is a combination of the names of the classes that participate in the association and a keyword that describes the nature of the association. This keyword (which generally is a verb) helps distinguish between the different associations that may exist among the same set of classes.

Property **`role`** specifies the association's role of the domain class that contains the `DAssoc` (i.e. the class that participates in the association with that role.)

Property **ascType** specifies the type of association, which is one of the followings: one-to-one, one-to-many, many-to-many. Similar to property `name`, the value of this property needs also be the same for the `DAssocs` in the same set.

Property **endType** specifies the type of association's end, which is either one or many.

Property **associate** specifies the opposite end of an association. The type of this property is another meta-attribute called `Associate`, which has four properties: `type`, `cardMin`, `cardMax`, `determinant`. Property **type** specifies the associated domain class of the opposite end. In Figure 3, the value of this property is written using the Java's class notation, which is consisted of a class name followed by the keyword extension `.class`. Properties **cardMin** and **cardMax** together specify the exact cardinality constraint of the associate's end. In Figure 3, the values of these properties are self-explanatory, except for those defined for the link attribute `Student.enrolments`. Here, we are able to specify a more fine-grained (and more realistic) cardinality range of [0,30], compared to the general range [M,N] implied by the diagram. Last but not least, property **determinant** specifies whether or not the associated domain class is the determinant of the association (i.e. a strong entity). The value of this property is `true` only for 1-1 associations whose one end is a strong entity. In Figure 3, `Student` is the determinant of the association between it and `City`.

Property **dependsOn** specifies whether or not the domain class at this end depends on that of the opposite end. If set to `true`, the domain objects of the domain class at this end can only exist when it is associated to a domain object of the class at the opposite end. Note that this notion of dependency differs from that entailed by property `determinant` above in that it may apply to any type of association (not just one-one). In Figure 3, for instance, we have `Enrolment` depends on `Student`.

Listings 2 and 3 below show the Java code snippets of the two domain classes `City`, `Student`, which realise the associations mentioned above. Listing 1 shown earlier contains the Java code definition of the domain class `SClass` that realises the association with `Student`.

```
@DClass(schema="courseman")
public class Student {
  @DAttr(name="id",id=true,auto=true,type=Type.String,length=6,
       mutable = false,optional=false)
  private String id;
  private static int idCounter = 0;

  @DAttr(name="name",type=Type.String,length=30,
       optional=false)
  private String name;

  @DAttr(name="address",type=Type.Domain,length=20,optional=true)
  @DAssoc(name="student-has-city",role="student",
       type=AssocType.One2One,endType=AssocEndType.One,
```

```
      associate=@Associate(type=City.class,cardMin=1,cardMax=1))
  private City address;

  @DAttr(name="sclass",type=Type.Domain,length = 6)
  @DAssoc(name="class-has-student",role="student",
      type=AssocType.One2Many,endType=AssocEndType.Many,
      associate=@Associate(type=SClass.class,cardMin=1,cardMax=1))
  private SClass sclass;
}
```

Listing 2: Java specification of some domain attributes and two associations of `Student`

```
@DClass(schema="courseman")
public class City {
  @DAttr(name="id",id=true,auto=true,type=Type.Integer,length=3,
       mutable=false,optional=false)
  private int id;
  private static int idCounter;

  @DAttr(name="name",type=Type.String,length=20,mutable=false,
      optional=false)
  private String name;

  @DAttr(name="student",type=Type.Domain,optional=true,
      serialisable=false)
  @DAssoc(name="student-has-city",role="city",
    type=AssocType.One2One, endType=AssocEndType.One,

associate=@Associate(type=Student.class,cardMin=1,cardMax=1,determinant=true))
  private Student student;
}
```

Listing 3: Java specification of the domain attributes and one association of `City`

4.5. Operations

Once the foundational structure of a domain class has been constructed using the meta-attributes, one then proceed to define the interface operations of the class. Here, the domain attributes together with their meta-attributes are used to identify and specify the *essential* operations. These include operations in the following categories [9][3]: constructor, mutator (e.g. setter), observer (e.g getter), and default.

For example, Listing 4 below shows the Java code of the essential operations concerning the domain attributes of the domain class `SClass` shown in Listing 1. The complete code listing of this class is provided in [6].

```
import domainapp.model.meta.DOpt;
@DClass(schema="courseman")
public class SClass {

  // derived attributes
```

3 In addition, FIT students should refer to the relevant PPL lectures on this

```
  private int studentsCount;

  // create objects directly from code
  public SClass(String name) {
    this(null, name);
  }

  // create objects from data source
  private SClass(Integer id, String name) {
    this.id = nextID(id);
    this.name = name;
    this.students = new ArrayList();
    studentsCount = 0;
  }

  private static int nextID(Integer currID) {
    if (currID == null) {
      idCounter++;
      return idCounter;
    } else {
      int num = currID.intValue();
      if (num > idCounter)
        idCounter = num;

      return currID;
    }
  }

  public int getId() {
    return id;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setStudents(List<Student> students) {
    this.students = students;
  }

  public List<Student> getStudents() {
    return students;
  }

  @DOpt(type=DOpt.Type.LinkAdder)
  public boolean addStudent(Student s) {
    if (!this.students.contains(s)) {
      students.add(s);
    }

    // no other attributes changed
```

```
    return false;
  }

  @DOpt(type=DOpt.Type.LinkAdderNew)
  public boolean addNewStudent(Student s) {
    students.add(s);
    studentsCount++;

    // no other attributes changed
    return false;
  }

  @DOpt(type=DOpt.Type.LinkAdder)
  public boolean addStudent(List<Student> students) {
    for (Student s : students) {
      if (!this.students.contains(s)) {
        this.students.add(s);
      }
    }

    // no other attributes changed
    return false;
  }

  @DOpt(type=DOpt.Type.LinkAdderNew)
  public boolean addNewStudent(List<Student> students) {
    this.students.addAll(students);
    studentsCount += students.size();

    // no other attributes changed
    return false;
  }

  @DOpt(type=DOpt.Type.LinkRemover)
  public boolean removeStudent(Student s) {
    boolean removed = students.remove(s);

    if (removed) {
      studentsCount--;
    }

    // no other attributes changed
    return false;
  }
}
```

Listing 4: Some essential operations of the domain class `SClass`

The listing shows three constructors, one getter operation for the attribute `id`, a setter and a getter operation of the attribute `name` and five operations related to the domain attribute `students`. Among these five operations, the first two are getter and setter, while the last three are to add and remove `Student` object(s) that are associated to an `SClass`. These last three operations, which are specified with a helper meta-attribute named `DOpt`, are needed only for collection-typed attributes. We will

discuss these operations shortly below.

The type of a parameter used in each constructor must be an object-type (which include Java built-in wrapper classes). The three-argument constructor invokes a class operation named `nextID`, which generates the next auto-generated value for the `id` attribute. This operation in turn makes use of a class attribute called `idCounter` to keep the maximum id value assigned so far. We explain further in [6] on how to maintain the values of auto-generated attributes between subsequent runs of a software.

Attribute `id` does not have a setter operation because it has `DAttr.mutable=false` (i.e. the attribute is immutable). The constructor and setter operations do not include input validation logic. This is because input validation is performed by a separate component (called **data validator**), which is called upon by the tool to check the input arguments before these are passed into the operations.

The following subsections will discuss three special types of operations which apply to collection-typed attributes: link-adder, link-remover, and link-updater. It must be stressed, however, that these operations are *only needed if the association involving the concerned attribute is used by the software*. A typical example of such use is to display all the objects in the collection as part of the object form of the associated domain class.

Link-adder operation

This is one of the three types of operation that are specifically designed for maintaining the value of a collection-typed domain attribute of a class. The other two operation types will be discussed in the sub-sequent subsections.

There are two specific types of link-adder operation: one is specified with `@DOpt(type=LinkAdder)` and the other is with `@DOpt(type=LinkAdderNew)`. The first type (named ***link-adder***)is used to add an existing object (which is either already in the heap or freshly loaded from the data source) to the collection, while the second type (named ***link-adder-new***) is to add a new association link to an object (which is either an existing object or a newly created one). An example of the first type is the operation `addStudent` in Listing 4, while that of the second type is `addNewStudent`.

As shown in the listing, the above two operations differ in how they update the state of the current `SClass` object. The link-adder operation only adds the association link (if it is not already existed) to `SClass.students`. The link-adder opeation not only does this but also update the value of the derived attribute `SClass.studentCount`. The definition of this attribute is given in the listing. It is used to maintain the object count, the design concept of which is discussed in [6].

The link-adder-new operation in Listing 4 applies specifically to 1:M associations. For 1:1 associations, the specification of this operation is much simpler and involves naming the operation with the prefix

setNew. An example of this is shown in Listing 5, which is the link-adder-operation for the class City, named setNewStudent, which facilitates the 1:1 association between this class and Student. In this case, the implementation of this operation is similar to that of the setter operation setStudent. In general, however, this would include any extra statements needed to update the state of the current City object.

```
@DClass(schema="courseman")
public class City {

  @DOpt(type=DOpt.Type.LinkAdderNew)
  public void setNewStudent(Student student) {
    this.student = student;
    // do other updates here (if needed)
  }

  public void setStudent(Student student) {
    this.student = student;
  }
}
```

Listing 5: Java code snippets of domain class City showing how to specify a link-adder-new operation for its 1:1 association with Student

Link-remover operation

This is the second type of operation that applies to collection-typed domain attributes. This operation is specified with @DOpt(type=LinkRemover). Listing 4 shows a link-remover operation named removeStudent, which removes a Student object from SClass.students and updates studentCount accordingly.

Link-updater operation

Listing 6 below shows an example of a third type of operation that applies to collection-typed domain attributes. This operation, which is specified with @DOpt(type=LinkUpdater), is used to update the state of the current object when the state of an associated object is changed. In the listing, operation updateEnrolment updates Student.averageMark of the current Student object when an associated Enrolment object e is updated (possibly causing a change in its Enrolment.finalMark).

We explain further in [6] about operation updateEnrolment and, more generally, about the design issue concerning the maintenance of attribute Enrolment.finalMark.

```
@DClass(schema="courseman")
public class Student {

  @DOpt(type=DOpt.Type.LinkUpdater)
  public boolean updateEnrolment(Enrolment e)  throws IllegalStateException {
    // recompute average mark using just the affected enrolment
    double totalMark = averageMark * enrolmentCount;
```

```
    int oldFinalMark = e.getFinalMark(true);

    int diff = e.getFinalMark() - oldFinalMark;

    totalMark += diff;

    averageMark = totalMark / enrolmentCount;

    // no other attributes changed
    return true;
  }
}
```

Listing 6: Java code snippets of the domain class `Student`, showing a mutator operation named `updateEnrolment` used for updating the attribute `Student.averageMark`

4.6. Model Serialisation

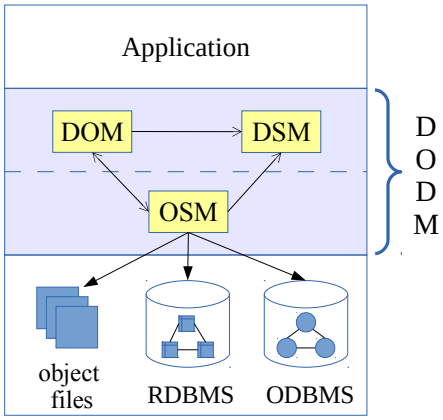

Figure 4: The Layered Design Architecture of DODM

A key advantage of our domain specification approach is that it allows us to develop automated techniques for serialising the domain objects into an external storage. The reason why this is possible is that the domain-specific design information (e.g. domain constraints) that are embedded in each domain class are sufficient for translating the domain class into a storage element (e.g. a relational table).

Our current implementation of the framework includes en extensible design for these techniques and an implementation of one technique that is for a Java-based RDBMS called JavaDB (a.k.a Apache Derby). We call the overall component that is responsible for managing the serialisation of the domain objects by the name **Domain-Driven Object Management (DODM)**. In this section, we will briefly explain the design of DODM.

4.6.1 Domain-Driven Object Management

Object serialisation requires the support for two storage-specific concepts, namely class store and store record, and a number of operations that are performed on these concepts. **Class store** generally represents the storage space that is used to store objects of a class. **Store record** represents the

serialised form of an object in a given class store. The operations that need to be performed with regards to these concepts are:

1. Create a class store for the objects of a class
2. Store an object into a class store as store record
3. Query a class store for objects that satisfy some criteria
4. Retrieve an object from the store record of a class store

The technology used to realise the above concepts may differ between storage providers. For example, RDBMS providers use their implementations of the relational model to implement the concepts. XML-based providers, on the other hand, use other structured-file technologies. To support these different types of technologies, we propose a layered DODM design shown in Figure 4.

As shown in the figure, DODM is consisted of three classes: `DSM` (domain schema manager), `DOM`, (domain object manager) and `OSM` (object storage manager). These classes that are distributed in two layers (whose separation is depicted by a dashed line in the figure). The top layer is the **schema management layer**, which consists of `DSM` and `DOM`. The bottom is the **object storage layer**, which is consisted of `OSM` and implementations of this class for the object technologies supported by the framework.

`DSM` defines a repository of all the domain classes of a software, while `DOM` maintains the object pool of each of these classes. `OSM` defines an interface to the object storage.

The dependencies shown in Figure 4 among the three classes are defined based on their roles in the DODM. Class `DSM` is used by the other two classes but is not dependent on them. Class `DOM` depends on both `DSM` and `OSM`. It uses `DSM` to look up domain-specific information about a class and its object, and uses `OSM` to store objects. Class `OSM` also depends on `DOM` and `DSM` for its operations. It uses `DSM` to look up domain-specific information about a class and its objects, and uses `DOM` to retrieve the objects that have already been loaded in the pools.

The reason for grouping the classes into two layers is to make it possible to not only support different object storage technologies but also to not depend on it. A software can be created with just the schema management layer, which manages its objects entirely in memory.

5. View

5.1. What Is A View?

A view is consisted of a number of different types of elements, including window and window elements (such as menu, data field, label, button, and the like). Our design aim in the framework is to automatically generate a module's view from the model and from the view configuration, which is specified in the module configuration. We will discuss in Section 7 the design of this configuration.

A module's view is an *object form* that presents the object state to the user and via which she can perform the module's operations on the objects. An **object form** is consisted of a set of pairs of data components and labels and an arrangement of these pairs on a user interface. A **data component** is a UI component that presents the value of a domain attribute. For example, Figure 5 shows the object forms of three COURSEMAN modules, namely $M_{Student}$, M_{SClass}, and $M_{Enrolment}$. The object forms are consisted of two types of data components: data field and data container. Two types of data field used in the figure are text field and spinner field. The object forms of the three modules use a two-column layout: the first column contains the labels and the second contains the components. The object form of a data container (e.g. that of the child module $M_{Enrolment}$ of $M_{Student}$) has a tabular layout.

All object forms are managed by another form, called the **main form** (the outer window in Figure 5). This form is the view of special module of a software called the **main module**. This module takes on the responsibility of managing the set of modules of the software.

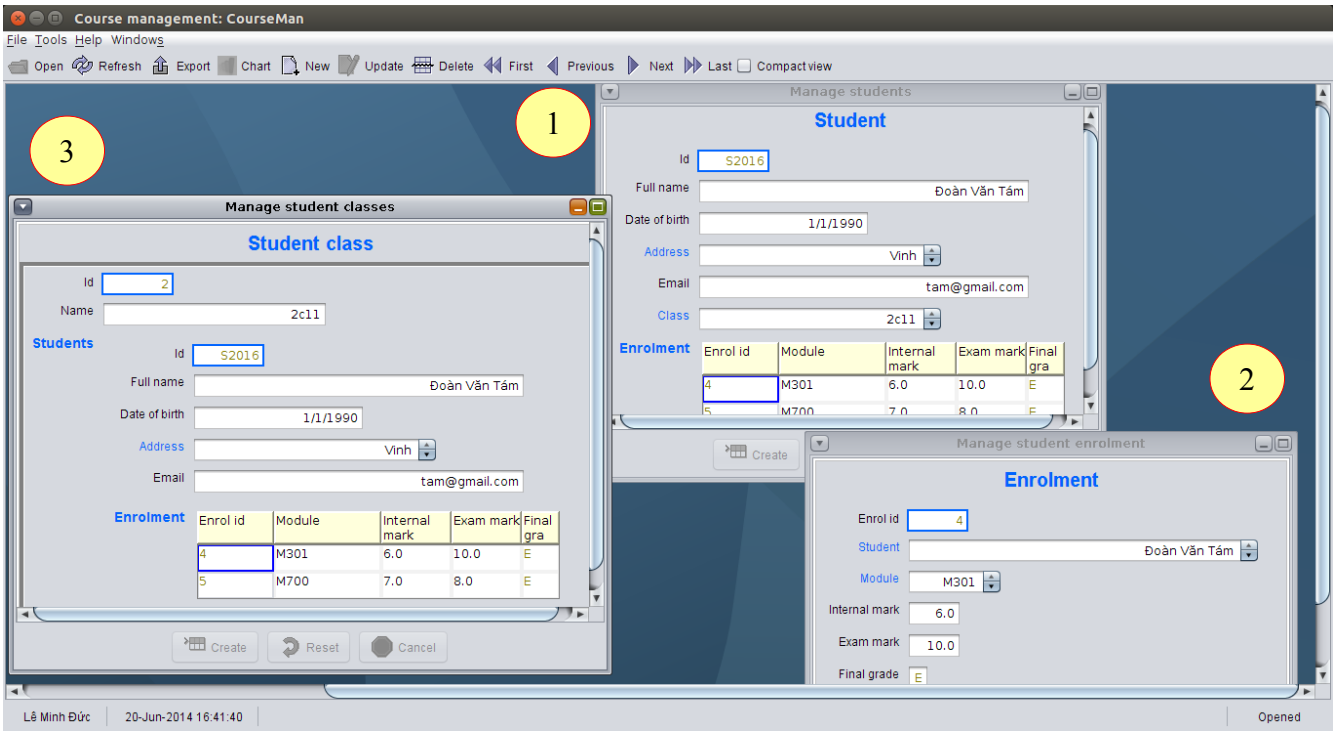

Figure 5: The prototypical views of three COURSEMAN modules:
(1) $M_{Student}$, (2) $M_{Enrolment}$, and (3) M_{SClass}

As shown in the figure, the main form consists of a menu bar and a tool bar, which define the commands through which the user can perform. There are two basic sets of commands, the event handling mechanism of which will be explained in Section 6. The first set, which contains the toolbar commands, are mapped to those of the object operations. These operations are performed against the active module (i.e. the module on which the user is currently working). The second set, which contain those defined in the `Tools` menu of the menu bar, are mapped to the initiation command of each module. When chosen, each command will initialise a module and display its view.

5.2. Design Rationale

To automatically generate the view requires making it possible for the designer to explicitly specify all the information needed to generate each view. These include such view-specific properties (including title, layout, etc.) and field-specific properties, i.e. the configuration of each data field that the view contains. These include such properties as size, font, colour, image icon, editability. To model these properties in the design, we introduce the concept of a region and represent it by a class named `Region`. A region represents any kind of named area on a GUI, including main window, sub-window, menu, data field, and even action button. In Java, a window is represented by the class `JFrame`, while a sub-window is represented by the class `JInternalFrame`. Details of the class `Region` will be explained in Sections 5.3-5.5.

The second design problem is how to represent each data field so that it can be re-used to create differerent types of view. To achieve this, we design each data field based on that of a corresponding Java Swing's component [10]; for example: a text-based data field is designed based on `JTextField`. The current Swing API provides an enriched set of components that are sufficient to build a sophisticated GUI. Further, Swing components are designed to support the customisation of the view properties mentioned above.

Last but not least is the design of the `View` class itself. This class is responsible for generating the module's view from the module's domain class and the configuration properties. To achieve this, it needs to extract from the domain class the set of domain attributes and the values of the meta-attributes that are attached to them. These will be used to determine the suitable data fields to use to represent these attributes and also the values of some of the configuration properties of these fields (e.g. if an attribute is imutable then the data field's editability must be set to `false`). Java provides reflection [11], which is used by class `View` to perform the extraction.

5.3. View Region

A **region** represents a named area on a GUI, which includes window, sub-window, menu, data field, and action button. This is modelled in the framework by the class named `Region`. The high-level design view of this class and its type hierarchy are shown in Figure 6.

Class `Region` defines attributes that represent view configuration properties. Among these are `name`, `label` and `displayClass`. Attribute `name` helps identify a region, while attribute `label` specifies the text that is used to label the region on the UI. For example, this attribute defines the text of a `JLabel` object that is used for a data field region. Last but not least, attribute `displayClass` specifies the Java class object that is used to render the associated region on the UI. Which class to use depends on the type of the region and the type of data that will be presented.

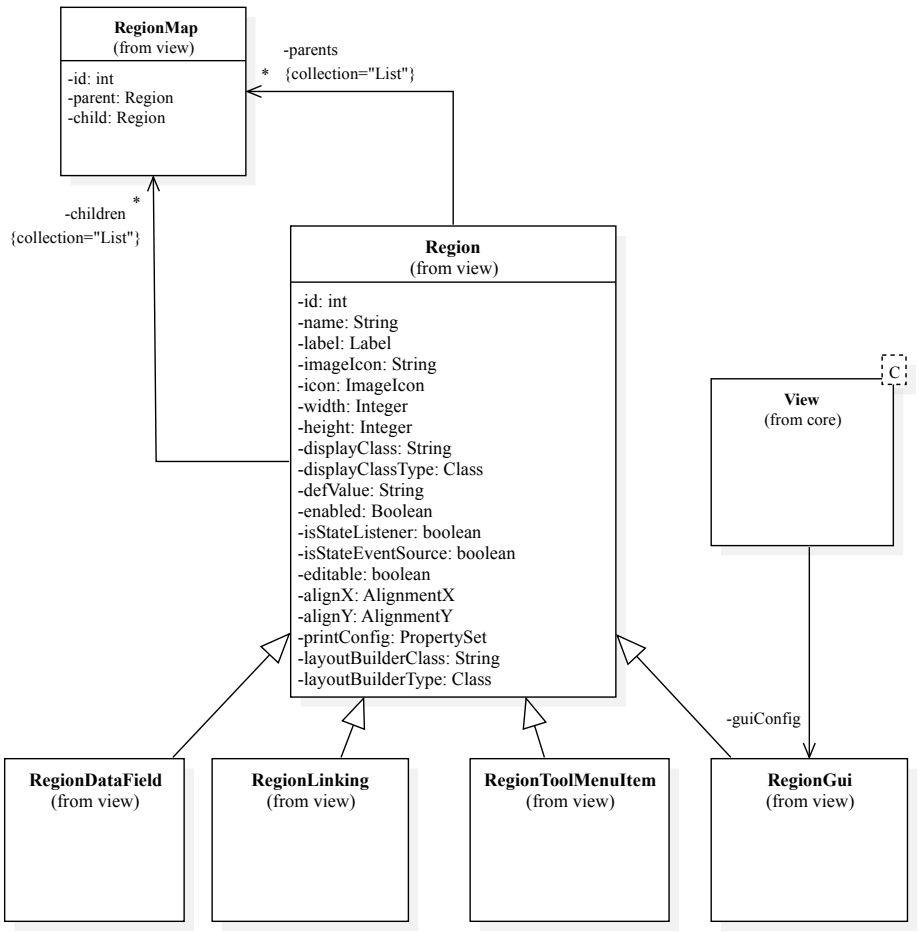

Figure 6: The design of view region

There are three region types that form the type hierarchy of `Region`, which we will now explain.

5.4. Region Types

Class `Region` has four sub-types: `RegionGui`, `RegionDataField`, `RegionLinking`, and `RegionToolMenuItem`. These sub-types are shown in Figure 6 and illustrated in Figure 8 for the CourseMan software.

Class `RegionGui` represents the region that spans the entire view of a module. For example, Figure 8 shows the `RegionGui` objects of three `CourseMan`'s modules. To ease reading, each region is denoted by the class name followed by the value of its attribute `name`. Further, when the context is clear the region object will be used to refer to the GUI component described by that object. Thus, the first `RegionGui` object in the figure is `RegionGui("CourseMan")`, which is a window-typed region that describes the main form of the software. This region is called the **main region** and has its `type` value set to `"Main"`. The second object, `RegionGui("Student")`, is a sub-window region that describes the object form of the module `Module<Student>`. The data fields of this form are created for the domain

attributes of the class `Student`. Similarly, the third object is the sub-window region that describes the object form of `Module<Enrolment>`.

Class `RegionDataField` represents a sub-region of a `RegionGui` that covers a data field and its label. A `RegionGui` is thus consisted of a set of `RegionDataFields`. Figure 8 highlights one example `RegionDataField`, namely `RegionDataField("id")`, which is mapped to the data field of the attribute `Student.id`. This region describes the display area containing the text data field and its label.

Class `RegionLinking` represents a link to the base configuration of the object form a child module that participates in a module containment. More detail of this will be given shortly in Subsection 5.5.

Class `RegionToolMenuItem` represents a sub-region of the main region that covers a menu item of the menu bar. The primary use of this region is to represent a menu item of each module that is displayed in the `Tools` menu. The names of these `RegionToolMenuItem` objects are taken from those of the modules' domain class. Figure 8 illustrates this by showing a `RegionToolMenuItem("Student")` object, which is mapped to a menu item of the `Tools` menu and that describes a command for initiating `Module(Student)` of the CourseMan software.

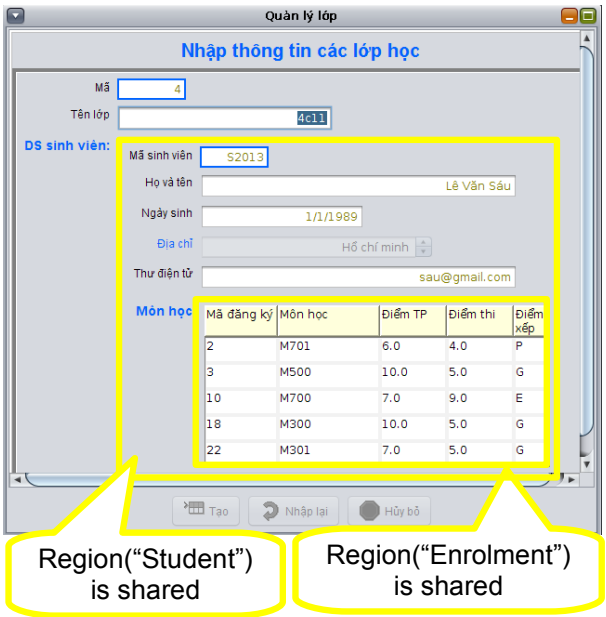

Figure 7: Two containing shared regions

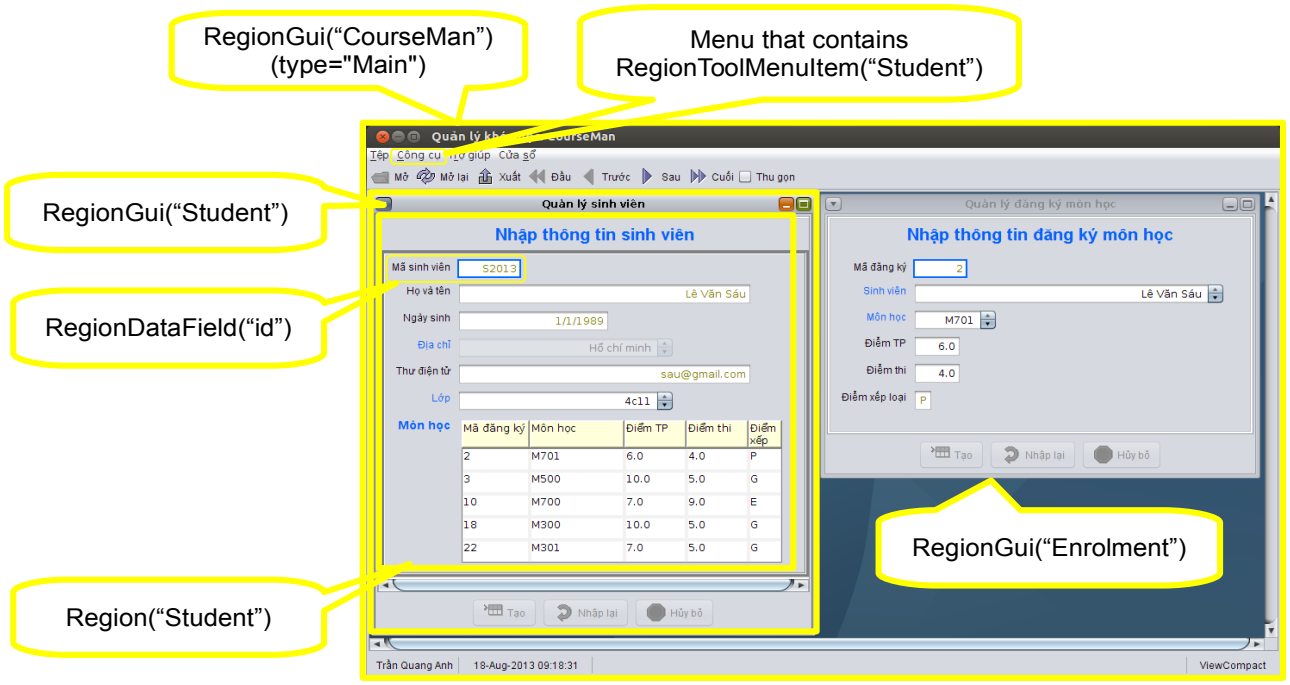

Figure 8: An illustration of the view configuration regions using the COURSEMAN software

5.5. View Containment

The module containment aspect of the architecture (see [3]) is realised by designing the view of each child module object as a child of that of the parent. This is realised in terms of two containment relationships: *component containment* and *region containment*.

5.5.1 Component Containment

Component containment is the containment relationship between an object form of a module and the set of child components, consisting of the form's data fields and the child (nested) object forms of the child modules, if any. Figure 8 illustrates this by showing how the data fields of the object form of `Module<Student>` are contained as children of this form. Further, an object form of `Module<Enrolment>` is also a child component of this form.

Component containment is supported by the many-to-many parent-child relationship between the Regions of the object forms and those of the data fields and of the child object forms. This relationship is modelled in Figure 6 by adding a class named `RegionMap` and linking this class to `Region` using two one-to-many associations. One association is to identify the parents of a child while the other is to identify the children of a parent.

Since all view components are Java Swing components, the component containment naturally follows that of the Swing's.

5.5.2 Region Containment

Region containment is a containment relationship between the regions that describe the modules's views. For efficiency, we design this containment in such a way that the region information concerning

the domain class of a module is shared among all objects of the module. Specifically, we encapsulate these information in a **shared region** object and use this as the basis to define the containment scope of each child module object that participates in a module containment.

Each child module references the shared region object via what we call a **link-type region**. All link-typed regions are represented by the sub-type `RegionLinking` of `Region`.

We design region containment as a reflexive, parent-child association of the `Region` class shown in Figure 6. Using this design, the common configuration properties, such as font and colour, are specified in the region object of the form (be it the main or object one). These properties are then used as the default for all the child regions, including those of the data fields and of the object forms of any child modules. Of course, these regions may also specify their own values of the properties, which are merged with the default values to give the actual values.

The parent-child association is also used to implement the link-type regions mentioned above. Specifically, link to a shared region object is specified by including this object as the child of a `RegionLinking` object. This link is then used as a bridge to load the data field configurations of the shared region.

Figure 7 shows the object form configured by `RegionGui("SClass")` to demonstrate both shared region and region containment. The figure has two shared region objects. The first object, `Region("Student")`, encapsulates domain-specific configuration of the class `Student` (of `Module<Student>`). The second object, `Region("Enrolment")`, encapsulates the configuration of the domain class `Enrolment`.

Object `Region("Student")` was used in Figure 8 to configure the object form `RegionGui("Student")`. It is now used in Figure 7 to configure the nested object form of an object of `Module<Student>` that is the child of the `Module<SClass>` object shown in the figure. This nested object form presents data about the `Student` objects that are members of a given `SClass` object.

Similarly, object `Region("Enrolment")` was used in Figure 8 to configure the (table-layout) object form `RegionGui("Enrolment")`. It is now used in Figure 7 to configure the nested object form of an object of `Module<Enrolment>` that is the child of the `Module<Student>` object mentioned above. This form presents data about the `Enrolment` objects of a `Student` object that is a member of a given `SClass` object.

5.6. Class `View`

Class `View` is responsible for generating both the object form and the main form. It takes as input a `RegionGui` object that specifies the view configuration. The association to this `RegionGui` object is shown in Figure 6. The navigable end of this association named `guiConfig` is realised by the attribute

`View.guiConfig`.

5.7. Data Container

The component that renders the display area of the object form that contains the data fields is called **data container**. This component is represented by the interface named `JDataContainer`, for which two concrete sub-types are defined: `DefaultPanel` and `JDataTable`. `JDataTable` represents all data containers that use a table-like layout to organise the data fields. `DefaultPanel` represents all other kinds of data containers.

The above interface and its two sub-types are shown in the lower left of Figure 9.

5.8. Form containment

Form containment refers to the containment of the object forms within the main form (discussed earlier). This is realised by a reflexive, parent-child association of the `View` class in Figure 1. This association is navigable at the parent end, which results in the inclusion of the attribute `View.parentGUI` in the design. This attribute is initialised to the argument typed `View` of the constructor operation of the class. The value of this argument is the `View` object of the main form.

Note that form containment is a type of component containment but is not necessarily the same as region containment (both were described earlier in Section 5.5). As such, the `RegionGui` object of the main form does not need to be the parent of those of the object forms that it contain.

5.9. View Generation

In this section, we will describe the algorithm that we use to automatically generate the view. This algorithm is implemented in the method `createGUI` of the class `View` which is invoked via the `Controller`.

Listings 7-9 are the pseudocodes of three main procedures that form the body of the algorithm. Listing 7 is the pseudocode of the main function, `createDataContainer`, which generates a `JDataContainer` object for a given object form. This function calls either the function `createPanelContainer` (Listing 8) or the function `createTableContainer` (Listing 9) depending on the type of the container class (*D*).

All three functions use the same input parameters. The first parameter is a module object (*M*) for whose object form the generated data container is to be used. The second and third parameters apply if *M* is a child module. The second parameter is the containment scope (ζ) of *M* w.r.t the parent module, while the third parameter is the data container of the parent module (of whom the generated data container is to be added as a child component). If *M* is not a child module then $\zeta = \oslash$ and *T* is not defined. The fourth parameter is the actual display class of the data container to be created. It is a sub-type of one of the two data container types discussed in Subsection 5.7.

Table 1 lists the utility functions used in the pseudocodes.

Functions	**Descriptions**
`isTyped(Class` *x*`, Class` *y*`)` → `boolean`	return `true` if the first class parameter is assignment-compatible to the second class parameter
`lookUpDataFieldRegion(` `Class` *c*`, Attribute` *a*`)` → `RegionDataField`	look up and return the child `RegionDataField` object of the object `Region<`*c*`>` that is mapped to the domain attribute *a* of *c*
`type(Attribute` *a*`)` → `Class`	return the declared type of a domain attribute *a*

Table 1: Utility functions used in the algorithm

5.9.1 Function `createDataContainer`

```
1 Input:
2     M: Module
3     ζ: containment scope
4     T: parent container
5     D: container class
6 Output: JDataContainer
7
8 if isTyped(D,DefaultPanel)
9     createPanelContainer(M,ζ,T,D)
10 else
11    createTableContainer(M,ζ,T,D)
12 end if
```

Listing 7: The pseudocode of procedure `createDataContainer`

5.9.2 Function `createPanelContainer`

A key property of this function is that it is recursive (line 23), which enables the creation of the data containers for all descendant modules that participate in a containment tree.

The specification of the containment scope is shown at line 20. It is specified as part of the domain constraint definition of the domain attribute (*a*) of the parent module's domain class (*C*), which links the data container of the parent (*P*) to that of the child (P_a). Refer to Section 4 for detail about the domain specification.

Note that because of the recursive call, the data container (*P*) needs to be populated with all the components first, before its layout can be created (line 29).

```
1 Input:
2     M: Module
3     ζ: containment scope
4     T: parent container
5     D: container class
6 Output: DefaultPanel
7
8 let C = domain class of M
9 let P = new_D(M,T)
10 if ζ = ∅
11    ζ = C.attributes
12 end if
13
14 for each attribute a of C s.t. a in ζ
15   let R_a = lookUpDataFieldRegion(C,a)
16   if a is normal attribute
17     let df = data field created from a, R_a
18     add df to P
19   else
20      let A = domain class referred to by type(a)
21      let ζ_a = containment scope of a
22      let M_a = an object of Module<A> whose containment scope (w.r.t M) is ζ_a
23      let D_a = R_a.displayClass
24      let P_a = createDataContainer(M_a,ζ_a,P,D_a)
25      add P_a to P
26    end if
27 end for
28
29 create layout for P
30
31 if T is defined
32    add P to T
33 end if
34
35 return P
```

Listing 8: The pseudocode of procedure `createPanelContainer`

5.9.3 Function `createTableContainer`

Unlike function `createDataContainer`, this function is not recursive. Thus, in our current version of the framework, a table-typed data container can only be used either for a normal module or for the leaf modules in a containment tree.

```
Input:
  M: Module
  ζ: containment scope
  T: parent container
  D: container class
Output: JDataTable

let C = domain class of M
let P = new_D(M,T)
if ζ = ∅
  ζ = C.attributes

for each attribute a of C s.t. a in ζ
  let R_a = lookUpDataFieldRegion(C,a)
  create table header of P from a, R_a
  let c = P's column that corresponds to this header
  let df = data field created from a, R_a
  set c.cellEditor = df
end for

if T is defined
  add P to T
end if

return P
end if
```

Listing 9: The pseudocode of procedure `createTableContainer`

6. Controller

In this section, we will explain the design of the class `Controller` with respect to its handling of five architectural responsibilities discussed in [2]: view selection, event handling, state change, (model) change notification and (view) update. But first, let us explain the design rationale

6.1. Design Features

6.1.1 Object-related Operations

We define three sets of essential object-related operations:

1. `createObject`, `deleteObject`, `updateObject:` create, delete, update the domain objects of a domain class
2. `createLink`, `deleteLink`: create, delete the association links between domain objects of associated domain classes
3. `search`: search for domain objects that satisfy some user-specified criteria

In designing the class for these operations, we had to choose between two alternative solutions:

1. via abstraction: assigning these operations to directly each domain class through the use of a shared interface or abstract super-class
2. via shared implementation: assigning these operations to a separate class and define generic implementations for them

There are pros and cons of each solution. The first solution is better in terms of performance but complicates the design of the domain classes and hence their maintenance (because the same set of operations need to be implemented in each domain class). The second solution incurs some performance penalty (compared to the first solution) due to the separation of the operations from the domain class (and thus Java reflection is required to perform the object operations), but is less troublesome for the developer to implement and maintain (the operations are implemented in one place and are shared among domain classes).

The second solution is chosen on the ground that Java's reflection is commonly used in designing software frameworks and that the performance penalty is tolerable given today's computing power.

Having picked this solution, however, the next problem that we faced was whether to define a new class for the operations or to use one of the existing architectural classes. We chose a hybrid solution – a new class that is attached to (nested inside) one of the architectural classes. The new class is named `DataController` (naturally due to the fact that its operations concerning the manipulation of the domain objects), and is designed as an inner class of class `Controller`. There are two reasons why we chose this class. The first reason is because the object-related operations are invoked by the event handling mechanism of the controller (this mechanism will be explained later in this section). The second and a more important reason is to do with the support for module containment, which we will explain shortly below – the `DataController` objects of the same module are created and managed by the `Controller` object of the module.

Figure 9 below shows the essential design of the `Controller` and `DataController` class.

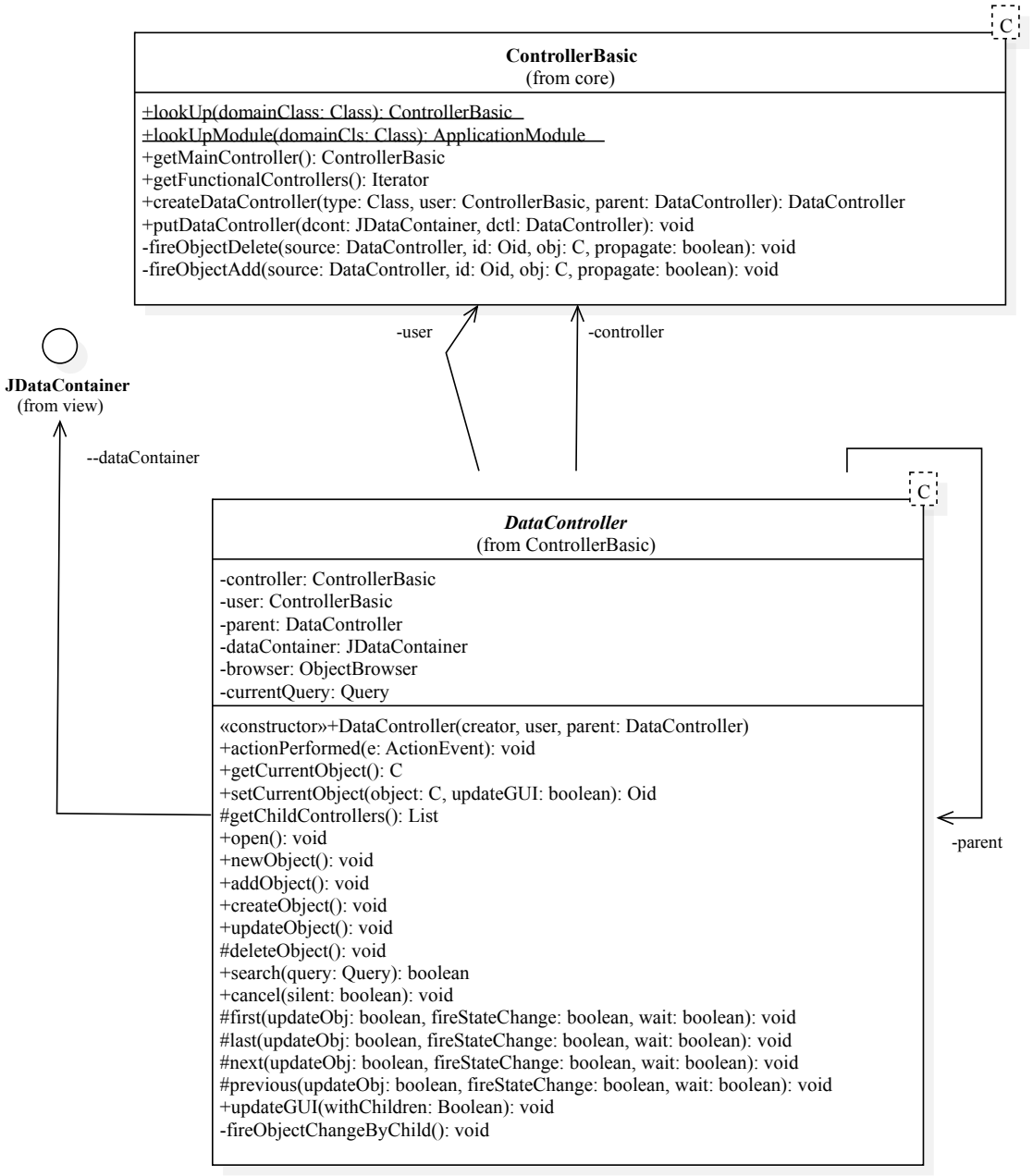

Figure 9: Designing object-oriented operations using class `DataController` and `Controller`

6.1.2 Controller Containment

As explained in Section 5, module containment was first realised in the view design by the concept of view containment. To realise this in the controller, we make two design decisions.

First, we associate `DataController` to the data container class (`JDataContainer`) and assign `Controller` the responsibility of a factory of `DataController` objects. More specifically, for each module object there is one `JDataContainer` object created for its object form and, correspondingly,

one `DataController` object created for handling the object-related operations that are performed on this form. The former is created as part of the creation of the object form, which as explained in Section 3.5 is performed by the `Controller` object of the module. The `Controller` class has a map-typed attribute called `dataControllerMap` (shown in Figure 1) that is used to record all pairs `<JDataContainer,DataController>` that it creates.

Second, we create a reflexive, parent-child association for class `DataController`, which realises the parent-child relationship between the parent and child modules.

6.1.3 Primary Data Controller

The first top-level `DataController` of a module is called the **primary data controller** of that module. This controller is used to perform object operations in response to a message exchange that is initiated from an associated module. We will explain this scenario in Section 6.5.

6.1.4 Data Controller Types

To support the different types of data containers explained in Section 5.7, we create a type hierarchy data controllers shown in Figure 10. Each sub-type of `DataController` specialises in handling a corresponding data container type. In particular, `DataPanelController` is used for `DefaultPanel` container while `TableDataController` is used for `JDataTable` container.

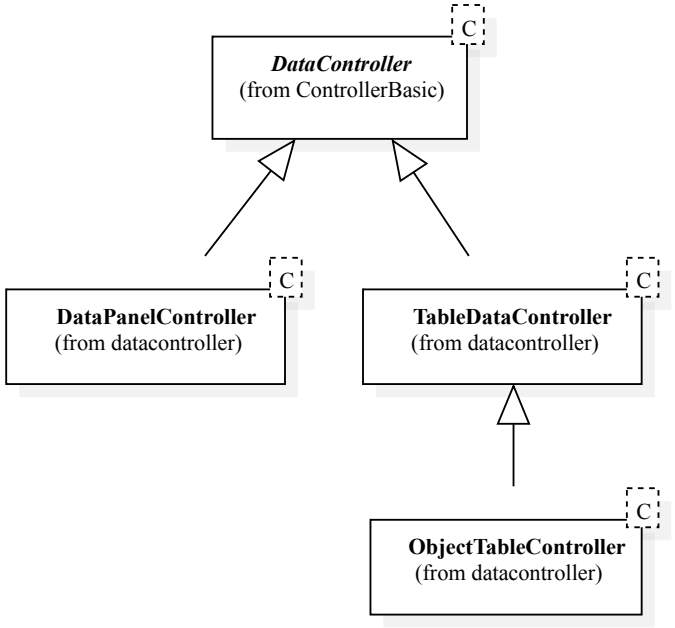

Figure 10: Data controller type hierarchy

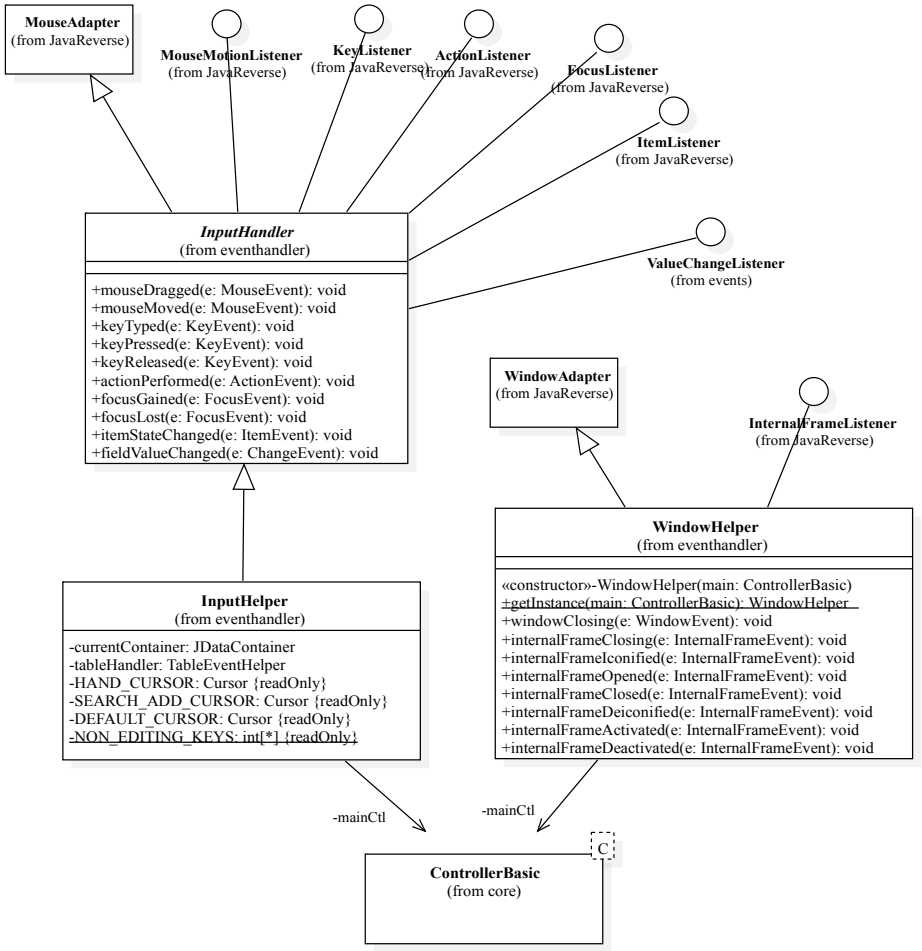

Figure 11: Module-related event handling

6.2. View Selection

View selection is straight-forwardly handled in the framework by the `Controller` object of each module.

6.3. Event Handling

An **event** is raised as the result of a user action that was performed on a UI component of a view. The event is mapped to an operation on some object, which is subsequently performed to yield result. The event handling role of the `Controller` is to make this mapping possible. More specifically, it maps maps each event to one of the functions described in Section 6.1.

6.3.1 Object-Related Event Handling

When user performs an object-related command from the tool bar of the main form the command is mapped, via the event handling method `DataController.actionPerformed` (shown in Figure 9), to an object-related operation to be performed on the `DataController` of the active object form. This operation is then performed and the result (if any) is presented to the user.

6.3.2 Module-Related Event Handling

Figure 11 shows the classes involved in handling the menu bar commands and the form-related actions.

6.4. State Change

When a state change occurs, the `Controller` updates the model using one of the `delete` operations and the `updateObject` operation.

6.5. Model Change Notification

When an object-related operation is performed on the object form of a module (the **source module**) results in a change in the state of a domain object of another module (the **target module**), which may or may not be the same as the source module, the `DataController` of the source module informs that of the target module to update the state of the object.

To realise this in the design, we differentiate between two types of message exchange: (1) inter-module exchange and (2) intra-module exchange. **Inter-module exchange** involves sending a change message to some `DataController`(s) of a different module, while in an **intra-module exchange** the message is sent to some `DataController`(s) of the same module.

Inter-module exchange is in turn classified into two types: parent-child exchange and associate exchange. **Parent-child exchange** is an exchange from a parent module to some of its child modules, or vice versa from one of these child modules to the parent. This is a special case of **associate exchange**, which is the exchange between associated modules (i.e. associations exist between the domain classes of these modules). Parent-child exchange is supported in the design using the parent-child association of `DataController` (explained previously). Associate exchange is realised in our design using controller look-up. More specifically, the `DataController` of the source module uses its `Controller` to request the main `Controller` to look up the `Controller` of target module. It then invokes the operation corresponding to the message upon the primary `DataController` of this.

Intra-module exchange is realised by taking advantage of the recording of the `DataController` objects of module in the `Controller` object (attribute `Controller.dataControllerMap`). We update the `Controller` class to add two event firing operations: `fireObjectAdd` and `fireObjectDelete`. The first operation is invoked when a new domain object is created, the second operation is invoked when an existing domain object is deleted. Both operations use the attribute `Controller.dataControllerMap` to update the `DataControllers` of interest.

6.5.1 Example: CourseMan

To illustrate, let us suppose the CourseMan software is now extended to record the GPA (Graduate Point Average) as follows. First, class `Enrolment` is extended with a new derived domain attribute named `final mark`, whose value is computed as the weighted average of the `internal mark` and the `exam mark`. Second, class `Student` is extended with a new derived domain attribute named `gpa`, which

is computed as the average of `Enrolment.final mark` of all the enrolled modules of a student.

Then consider the scenario in which the user changes `Enrolment:internal mark` of a given enrolment. This will cause a change in `Enrolment.final mark` of that enrolment and consequently a parent-child exchange to update `Student.gpa` of the concerned student. In this exchange, the child module is $M_{Enrolment}$ and the parent module is $M_{Student}$. The `Controller` of $M_{Enrolment}$ will raise a state change event to inform the `Controller` of $M_{Student}$ about the change. This `Controller` in turn processes the change by updating the value of `Student.gpa` of the concerned student accordingly.

For the sake of the example, let us suppose further that class `SClass` is extended with an attribute named `SClass:averageGpa`, which is the average of all the `Student.gpa` of all the `Students` of a particular `SClass`. Then the above change in `Student:gpa` will result in the change in `SClass.averageGpa` of the `SClass` of the concerned `Student`. To make this happen, an associate exchange is raised in which the `Controller` of the afore-mentioned $M_{Student}$ raises a state change event to inform the `Controller` of M_{SClass}. This `Controller` will process the change by recomputing the average GPA and updating the attribute `SClass.averageGpa` with it.

6.6. View update

The `DataController` has an operation named `updateGUI`, which is specifically designed for updating the `JDataContainer` associated with it. If the cause of the update is a change in the state of the domain object, then this operation updates the data container to show the new state of the domain object. For instance, consider the extended COURSEMAN requirement above, after updating the value of `Student:gpa` of the concerned student, the `DataController` of $M_{Student}$ invokes operation `updateGUI` to update the data container to show the new `Student:gpa` value.

7. Software

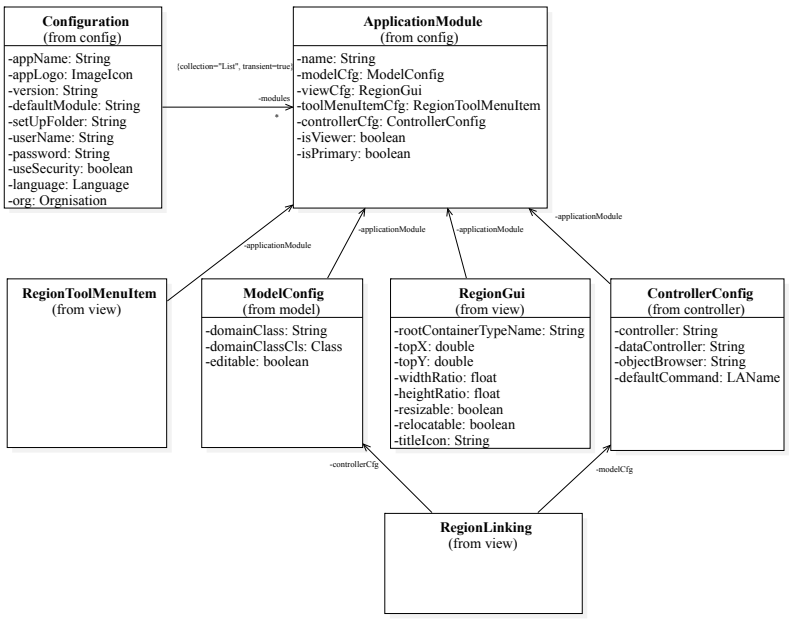

Figure 12: The design of software configuration

To model the concept of a software as a set of modules, we use a class named `Configuration`, which encapsulates the software settings and an aggregation association between this class and the `ApplicationModule` class. This class was introduced in Section 3.1 as a representation of the module configuration. Figure 12 is the essential design diagram of the software configuration. We briefly explain below the essential design features of each of the classes in the figure.

7.1. Class `Configuration`

This includes attributes that describe general technical information about the software and about the owner organisation. For example, it includes such attributes as software name, version, language, and the organisation for which the software is developed.

A `Configuration` is composed of one or more `Application Module`(s).

7.2. Class `ApplicationModule`

This class includes attributes that describe a module. Among these are the module name, title, and (displayed) image icon.

An `Application Module` is composed of one `ModelConfig`, one `RegionGui`, and one `ControllerConfig`. These objects specify the configurations of the three components that make up the module.

7.3. Class `ModelConfig`

This class includes an attribute named `domainClass`, that specifies the FQN of the domain class a

module.

7.4. Class `RegionGui`

Class `RegionGui` (discussed in Section 5) is used to model configuration of the view component of a module. It contains an attribute named `displayClass`, that specifies the FQN of the actual display component class used to render the display of the associated view. The designer can use one of the classes provided by the framework or a customised class as value of this attribute.

7.5. Class `ControllerConfig`

This class includes an attribute named `controllerClass`, which specifies the FQN of the `Controller` class of a module. The default value of the attribute `Controller.controllerClass` is the class `Controller`. However, sub-types of this class, which contains customised features, can also be used.

7.6. Class `RegionLinking`

As discussed in Section 5.5, this class represents the configuration of the link-typed region. The design in Figure 12 shows that the link-typed region can be customised with a suitable `ModelConfig` and `ControllerConfig`.

8. Software Configuration Method

Ideally, the software configuration design presented in the previous section is used as the basis for developing a configuration module (created in the framework) that provides the software designer with a GUI for creating the modules of a software. We are investigating the development of this module as part of our future work.

As of this writing, we are using an alternative method to programmatically writing the configuration directly in Java. The details of this method are presented in [12]. The basic idea is to use a module configuration class (MCC) as the skeleton on which to define the module elements. There is one MCC for each module. Attached to this MCC are a set of Java annotations, each of which is used to specify values for the objects of a configuration class.

9. A Light-weight Software Development Method

It can be seen from the discussion so far that our framework helps significantly reduce the effort needed to develop a software by making the most of the domain-driven design features to automatically generate the software code. In this section, we will briefly explain a light-weight software development method that uses the framework.

The method is light-weight because it consists of two main activities:

- develop the **domain classes**
- configure the **modules**

Briefly, the developer first develops the domain classes using specification described in Section 4.

Next, the developer configures the software modules using the domain classes and the `Controller` and `View` classes provided by the framework. If customised features of a framework's class are needed for some module then these features can be developed in a sub-type of this class.

Once the developer has completed the above activities, the framework will automatically generate the software and its modules.

There are three modes of software generation supported by our framework: domain modelling prototype generation, development prototype generation, and production software generation. The first two modes are used to generate software prototypes. The third mode is used to generate the final software product. We briefly explain below these three phases.

9.1. Domain Modelling Prototype Generation

In this mode, the software prototype is generated from just the domain model. This prototype is primarily used for gathering the domain requirements and discussing how to realise these requirements in the domain model.

This mode does not require any modules to be created. It also does not require any special set-up. In particular, the domain model is automatically serialised to a built-in Java database. Please refer to [6] for a detailed discussion of how to use this mode.

9.2. Development Prototype Generation

In this mode, the software prototype is generated from the software modules that are currently being developed. The resulted prototype is used to finalise the user requirements concerning each software module.

9.3. Production Software Generation

In this mode, the final software is generated from the software modules using a set-up process. This set-up process is implemented as part of a special system module.

10. Related Work

There are different approaches for realising the MVC architecture for use in practice. Of particular relevance to our work are those that produce a concrete software framework. Two popular frameworks, Spring [13] and Struts [14], propose to use a conceptual architecture model in which the view is passive and the controller is used as a mediator between the model and the view.

Our framework differs from these frameworks in the following aspects. **First**, our framework's architecture makes explicit support for modules and uses a tree-based design to compose modules. **Second**, although these frameworks also use meta-attributes to specify the domain-specific design information of the model, the meta-attributes that they use do not support domain constraint properties like ours. **Third**, our framework supports automatic generation of the view, while the other two frameworks still require developers to write the view's code. **Fourth**, our framework is designed to be a

generic generic framework (although the prototype implementation currently only supports desktop software). In contrast, the other two frameworks are specifically designed for web-based software.

11. Conclusion

In this paper, we discussed the design and implementation of a domain-driven software framework based on a tree-based, domain-driven software architecture that we recently proposed. We explained the design of each of the architectural component (model, view, and controller) and discussed how these are put together to form a module and how the modules are put together to form a software. Finally, we described a ligh-weight software development method using the framework, that helps significantly reduce the development effort needed to develop a software.

Our plan for future work is to improve and extend both the design and the implementation in the following aspects:

- **view**: support other types of view (e.g. web-based)
- **platform**: extend the design to support other software platforms (e.g. web)
- **tool support**: develop a graphical design tool to ease the task of designing the domain class model and configuring the software modules

References

[1] G. E. Krasner and S. T. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *J. Object-Oriented Program.*, vol. 1, no. 3, pp. 26–49, 1988.

[2] I. Sommerville, *Software Engineering*, 9th ed. Pearson, 2011.

[3] D. M. Le, "A Tree-Based, Domain-Oriented Software Architecture for Interactive Object-Oriented Applications," in *Proc. 7th Int. Conf. Knowledge and Systems Engineering (KSE)*, 2015, pp. 19–24.

[4] D. M. Le, "A Domain-Oriented, Java Specification Language," in *Proc. 7th Int. Conf. Knowledge and Systems Engineering (KSE)*, 2015, pp. 25–30.

[5] D. M. Le, D.-H. Dang, and V.-H. Nguyen, "Domain-Driven Design Using Meta-Attributes: A DSL-Based Approach," in *Proc. 8th Int. Conf. Knowledge and Systems Engineering (KSE)*, 2016, pp. 67–72.

[6] D. M. Le, "DomainAppTool v5.0: A Domain-Driven Software Development Tool," Hanoi University, Hanoi, 2017.

[7] V. Cepa and S. Kloppenburg, "Representing explicit attributes in UML," in *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

[8] J. A. Hoffer, J. George, and J. A. Valacich, *Modern Systems Analysis and Design*, 7 edition. Boston: Prentice Hall, 2013.

[9] B. Liskov and J. Guttag, *Program development in Java: abstraction, specification, and object-oriented design*. Pearson Education, 2000.

[10] Oracle, "Trail: Creating a GUI With JFC/Swing (The Java™ Tutorials)," 2015. [Online]. Available: http://docs.oracle.com/javase/tutorial/uiswing/index.html. [Accessed: 12-Jun-2017].

[11] Oracle, "Trail: The Reflection API (The Java™ Tutorials)," 2015. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/index.html. [Accessed: 12-Jun-2017].

[12] D. M. Le, D.-H. Dang, and V.-H. Nguyen, "Generative Software Module Development: A Domain-Driven Design Perspective," in *(Submitted to) Proc. 9th Int. Conf. Knowledge and*

Systems Engineering (KSE), 2017.
[13] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, and R. Harrop, “Spring Framework Reference Documentation: Chapter 17 Web MVC framework,” 2016. [Online]. Available: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html. [Accessed: 12-Jun-2017].
[14] Apache-S.F, “Apache Struts: Key technologies primer,” 2016. [Online]. Available: http://struts.apache.org/primer.html#mvc. [Accessed: 12-Jun-2017].

Appendix 1 Detailed Design Class Diagram of the Core Classes

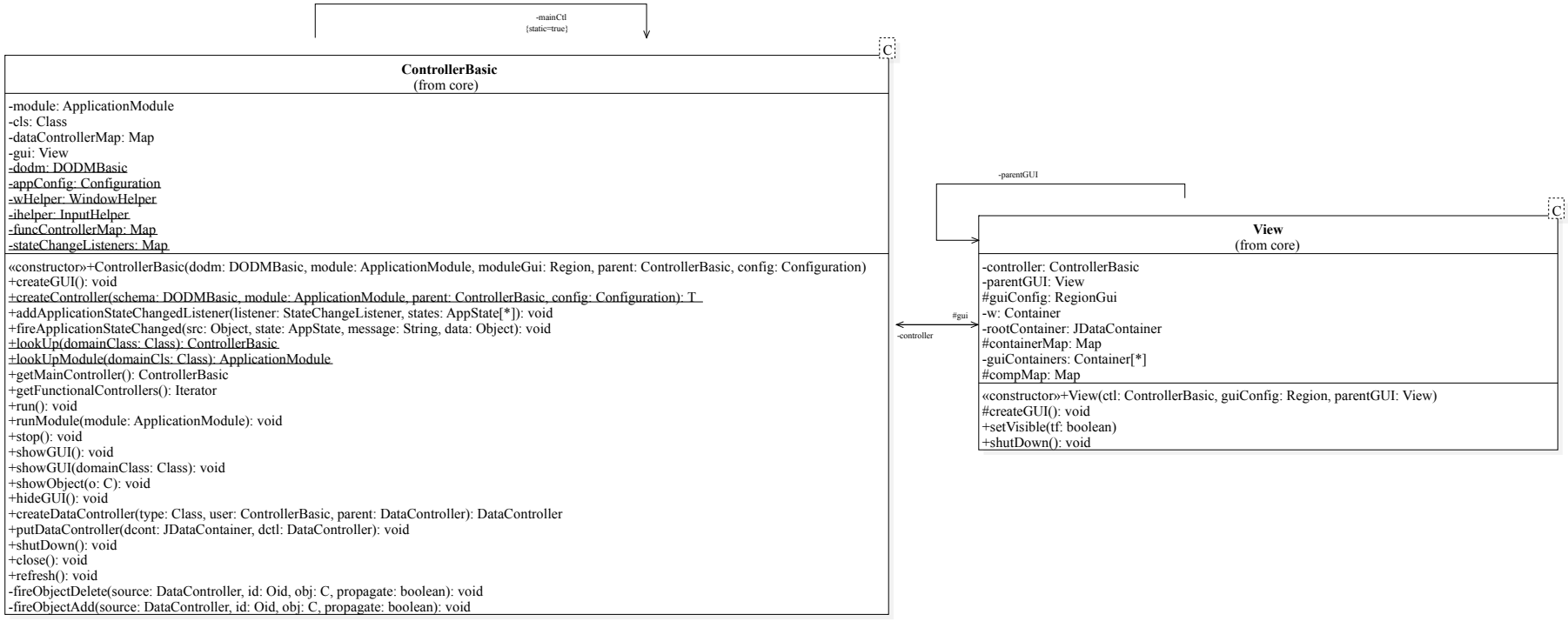

Figure 13: Detailed design class diagram of the core classes