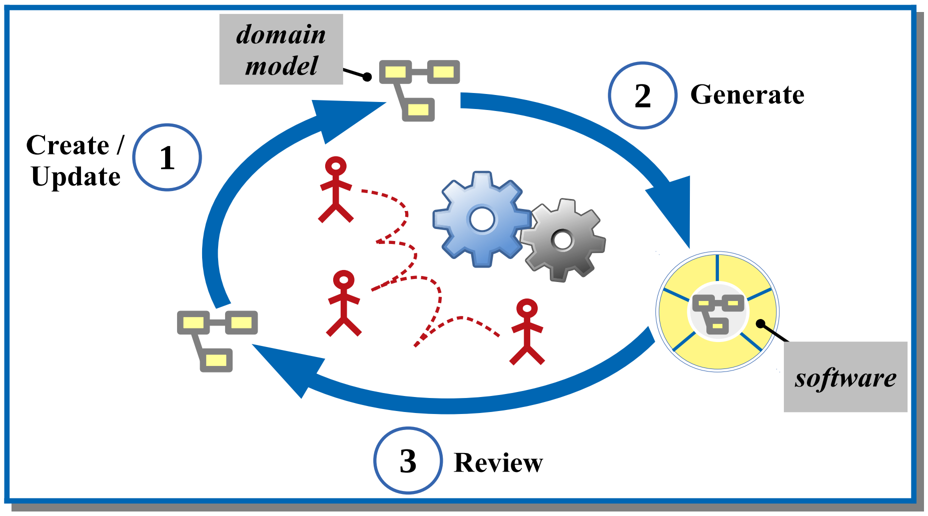
Object-Oriented Domain-Driven Software Development with DomainAppTool

A Software Engineering CourseBook



Dr. Duc Minh Le

Version: 5.4

ĐẠI HỌC HÀNỘI

Hanoi, 2019-2021

Table of Contents

Glossary

DDD	Domain Driven Design
OOD	Object Oriented Development
GUI	Graphical User Interface
MVC	Model-View-Controller
IDE	Integrated Development Environment
API	Application Programming Interface
DDSDM	Domain-Driven Software Development Method

List of Tables

List of Figures

List of Code Listings

Preface

Object-oriented software development (OOD) [1]–[6] has been a major force in the industry for nearly fourty years. What make OOD particularly effective are its natural and modular modelling of the problem domain, using objects and their interactions. This helps ease problem solving and software maintenance. About twenty years after the first OOD's publication, Evans [7] formulated a domain-driven design (DDD) method for developing software. DDD offers a domain-specific outlook for software development methods like OOD. Although DDD is not exclusively for OOD, the two are a natural match. Conceptually, a *domain* is a specific area of interest to software development. In DDD, the domain requirements are analysed to construct a *domain model*. DDD considers this model to be the "heart" of software, because it is where the complexity lies. With this view, the DDD method gathers the best design practices for frequently-occurring domain modelling problems and defines them as patterns – "tools" that designers can use to solve the design problems for their own software.

The DDD method itself does not provide any tools for software developers to use. This leads to independent groups proposing tools for the method. Among the available DDD tools are Apache-IsIs [8], [9] (the successor of Naked Objects [10]) and OpenXava [11], [12]. The former is perhaps a model DDD tool, to which Evans has been contributing. The latter is a mature tool for developing real-world software projects.

In 2012, a foundation was laid by this book's author with the development of a domain-driven software framework named JDOMAINAPP [13]. This framework was originally conceived to be used only as a tool for teaching an object-oriented program development method in the Java programming language [14], [15]. This method, which is proposed in an authoritative book [4], was updated and adapted by this book's author to incorporate the DDD philosophy. The improved method was then used to teach two core software engineering course modules at the Faculty of IT (Hanoi University).

In 2014, the development of DomainAppTool, which is based on JDOMAINAPP, began when a need was realised for a user-friendly tool that can be used to quickly and interactively execute and test the domain model. This tool has been tried at the Faculty of IT in teaching another software engineering course module named Special Subject 2 (SS2) and proved useful.

We observe that the development of DomainAppTool and, more generally, the JDOMAINAPP framework continuously evolves. The tool is used not only to demonstrate the framework's capabilities but to quickly experiment with any new domain modelling ideas that would eventually be incorporated into the framework.

By the end of 2016, our development had reached a crucial milestone when three conference papers [16]–[19] were published with an aim to consolidate and formalise (to a certain extent) the core theories that underlie the framework and tool. Two papers [18], [19], in particular, were collaborative research works between this book's author and Dr. Duc-Hanh Dang and Dr. Ha-Viet Nguyen from the Department of Software Engineering (VNU University of Engineering and Technology). Among these, paper [18] was extended into a journal paper [20], which has recently been accepted for publication.

Apart from research, the jDomainApp framework has been used to develop real-world software projects. We believe that this is the only pathway to challenging the underlying software engineering theories and to truly testing and stretching the framework's capabilities. Most recently in 2017, we successfully applied jDomainApp framework in a university-funded project to develop a software named ProcessMan (Process Manager) for managing organisational processes at the Faculty of IT (Hanoi University).

This coursebook was written based on the results of the aforementioned research papers and using the latest (possibly unpublished) results that we have recently discovered. We hope that you will find the book and the DomainAppTool interesting and useful!

Goals

This book is suitable for advanced software engineering students who wish to apply their software engineering knowledge and skills to develop large and complex software. The book aims to equip students with advanced programming knowledge and skills in using software framework. Further, the book will provide a practical software engineering method that uses the framework to automate software development tasks.

Prerequisites

To make the most of this book, students should have already completed the following course modules (or equivalent):

1. *Object oriented programming using Java*: describes the structure of an OO program and a method for designing and coding such a program. The coding language is Java.
2. *Software engineering*: describes advanced OO programming concepts (needed to develop large OO programs) and an object-oriented software engineering method for developing software. In particular, students should be familiar with the class and use

case diagrams of the Unified Modelling Language [21] and have a basic understanding of the Model-View-Controller (MVC) architecture [5]. Further, students should understand the notion of iterative software development process [22], [23].

Approach

We chose Java as the programming language because it is "free" and a popular object oriented programming language. Further, Java programs are portable to most computing platforms. A unique feature of our approach is to apply Java to develop the entire DDD software. This includes first developing the domain model in Java (especially using its annotation feature) and then applying a Java software tool named DomainAppTool that we have developed to automatically generate the entire software from the domain model.

Methodologically, we first explain how to construct the domain model and the software in one go. This helps readers become familiar with the domain modelling language that we use. After that, we introduce an iterative software development method and use it to explain how to incrementally develop the domain model and the software in iterations.

Structure of the book

This book consists of 7 chapters, which we summarise as follows:

- *Chapter 1*: describes a threaded, practical software development case study that we use throughout the book to illustrate the concepts and method that we propose.

- *Chapter 2*: gives a high-level architectural description of the DomainAppTool and presents a brief tutorial on how to work with the software graphical user interface (GUI). The same GUI is used to manipulate objects of different types of domain classes. These include domain classes that model domain concepts and those that model reports.

- *Chapter 3*: provides a detailed technical description of how to construct a domain model using a set of annotations. We call these annotations meta-attributes as they add metadata to the model elements. A domain model consists of a set of domain classes and the associations between them.

- *Chapter 5*: explains how to use the DomainAppTool to automatically generate a software from the domain model. We also discuss the structure of the underlying database of the software and the software management tasks.

- *Chapter 6*: discusses an advanced modelling feature for reports. We consider reports as domain classes with some special features.

- *Chapter 7*: explains a software engineering method that we propose to use with DomainAppTool. Using this method, the developers can systematically and incrementally develop the domain model to match the functions that the software perform.

Exercises

To help practise the concepts introduced in the book, we provide a set of exercises at the end of each chapter. In addition, these exercises are collated into a separate workbook that is made available together with from this book. We also provide an instructor workbook that contains answers to and instruction guides for the exercises.

A model course syllabus

In this section, we describe a model course syllabus that adopts this book as the main textbook. This model syllabus is based on a syllabus that we have been using since 2015 at the Faculty of IT (Hanoi University) in teaching a software engineering course named “Special Subject 2” (SS2). We share the syllabus with both teachers and students, hoping that it will be useful for them in planning their teaching and studying of the DomainAppTool and the accompanied software development method. In the remainder of this section, we will refer to the model syllabus simply as “course syllabus”.

We adopt the Stanford’s syllabus template [24] to structure the course syllabus, but, for the sake of generalisation, only discuss here the key elements listed below. Appendix 3 provides a complete syllabus example that we use for the SS2 course.

- *Course description*: describes what the course is about.
- *Learning goals and objectives*: states the course goals and the specific knowledge and skills (objectives) that students are expected to achieve at the end of the course.
- *Readings and resources*: lists the mandatory reading and learning materials. These include the current coursebook and the DomainAppTool software.
- *Course policies and expectations*: states the assessment components of the course (including assignments, homeworks, exams, etc.).
- *Learning toolbox*: describes to the students the learning approach of the course.

- *Syllabus schedule (teaching plan)*: sequentially describes the course topics and their teaching plan.

Course description

This course provides students with a hands-on experience in building a real-world software using a modern software framework. Students will first study a software development tool named DomainAppTool, which is the developer interface of a Java software framework. Next, students will form into small groups to investigate and define a suitable software development problem. Each student group will then use DomainAppTool, together with the software engineering and IT-related knowledge and skills that they learnt in the prerequisite courses, to develop and evaluate a software solution for their problem. The groups will report their progresses and receive feedbacks from the class and the teaching staff in the weekly meetings.

Learning goals and objectives

The overall goals of the course are to learn advanced software engineering knowledge and skills required to become fluent in using a software development framework and to apply this framework to solve problems in a wide variety of real-world domains.

More specifically, at the end of this course students are expected to:

1. Understand a modern software framework and tool for developing software.
2. Understand the "hot" domains for software engineering, especially in the (local) geographical context of the course.
3. Apply a software framework and tool and its software development method to analyse, design, implement, and evaluate a software solution for a real-world problem domain.

Readings and resources

The main mandatory reading of this course is the current coursebook. Accompanying this book are the following resources:

- *DomainAppTool software*: the latest binary of the DomainAppTool software. It is available at `https://github.com/DDDNet/domainapptool` [25].
- *Case study*: the Java source and binary codes of the case study named CourseMan (explained later in Chapter 1). These resources are needed to run the examples given in this coursebook. The codes are available in the `examples` folder of the DomainAppTool's download page [25].

Course policies and expectations

Table 1: The core assessment components of the course

Assessment		Weight	Brief Description
Group software project	Project progress report	5%	Each group submits a project progress report every week and presents one progress report to the class. The maximum mark of the presentation: 8/10.
	Software	25%	The final software
	Technical report	10%	The final technical report describing the software
	Final exam	60%	Oral examination: each group demonstrates the software and is asked a set of questions about it.

The assessment components are defined based on a group software project that spans the entire course. Table 1 summarises the core assessment components of the course. In order to achieve these, students are expected to work diligently and collaboratively every week to develop the software.

The **project progress report** is the weekly assessment, which is designed to motivate the groups to put incremental effort into the project. However, only one progress report from each group is expected to be formally assessed. This involves the group presenting the report content to the class to receive feedbacks. The maximum mark for this assessment is 8/10, because the software is not expected to be finished at this time. This mark is chosen so that if a group performs well in both the progress report assessment and the final software and report (explained next) then they would still be able to (after rounding up) attain the maximum mark of 10.

The **software** and **technical report** are essential because they are two standard deliverables in every real-world software project. The report describes what the software is about and provides technical details concerning its construction.

Last but not least, we argue that **oral examination** should be used to formally check that each group did indeed perform the necessary works in order to produce the deliverables that they submitted. There are two main reasons for this choice of final assessment. First, each software solves a different problem and, thus, it is not possible to construct one set of pre-defined questions to quiz all groups. Second, teacher's observation is required in order to check the authenticity of software construction.

Guidelines for software problem investigation in Vietnam

Considering Vietnam as the local context of the course, the following paragraphs provide some guidelines for students to limit their software problem search. At the time of this writing, these guidelines describe a scope for the "hot" problem domains in Vietnam, of which IT students in general should at least have some basic understanding. Note that this scope is expected to evolve over time.

Vietnam is well known for the diversity and quality of its agricultural products and services. In recent years, the Vietnamese government has strongly been promoting the use of and investing in modern agricultural technology. The investment appears to start baring fruits. According to a recent study[1], in 2017 and for the first time, agricultural export had taken over oil export to become the top contributor to the Vietnam's economy. Vietnam's agricultural products are currently exported to 180 countries in the world. More recently, agriculture has been considered[2] one of the three core areas in Vietnam's strategy for the Industrial Revolution 4.0 ("Cách mạng công nghiệp" 4.0).

Students are asked to investigate and choose one or a group of related architectural products/services for the software development project. The aim is to construct a software that would help improve the effectiveness of various activities that are performed in the life cycle of the chosen products/services.

In addition to using the software framework listed in the teaching plan to develop the software, students need to propose at least one improvement feature for this framework.

Learning toolbox

Since this course is project-based and very practical in nature, students are advised of the following key learning tools that are used:

- *Team work*: students will spend most of the time working with other members of the group. Communication and contribution are key ingredients to effective team work.
- *Project management*: student groups should plan their projects early and commit to the plan. On the other hand, they should be aware of alternative routes for performing the project tasks.
- *Problem identification and analysis*: students are not given a problem! They need to find a real-world problem and solve it. There are guidelines to limit the problem scope.

1 see http://vietnamnews.vn/politics-laws/416671/agriculture-leads-all-exports.html

2 see http://nhandan.com.vn/nhandan.com.vn/kinhte/item/35358302-co-hoi-vang-tu-cuoc-cach-mang-cong-nghiep-4-0-ky-1.html

- *"Semi-supervised" learning*: a technical term that we borrow from machine learning [26], which in this context means students are expected to learn by themselves under some guidance from the teacher. More specifically, the teacher supervises the students learning about the software framework and how to apply it to solve a model software problem. The students are then expected to apply their "learning process" to solve a new (unspecified) problem.

Syllabus schedule

Weeks	Descriptions	Tutorials
1	**Introduction** to the module, topic list **DomainAppTool: Introduction** *Ref*: Chapters 1-2	Set up and run DomainAppTool
2	**DomainAppTool**: **Domain class development (1)** *Ref*: Chapters 3.1-3.4, 5	Domain class development exercises (1)
3	**DomainAppTool: Domain class development (2)** *Ref*: Chapters 3.5-3.6.2	Domain class development exercises (2)
4	**DomainAppTool: Domain class development (3)** Ref: Chapters 3.6.3-3.7	Domain class development exercises (3)
5	**DomainAppTool (4): Report class development** *Ref*: Chapter 6	Report class development exercises
6	**Problem researchdài** *Ref*: Teacher's guidelines, Internet-based search, software engineering and IT-related knowledge from previous modules	Problem research report
7	**Software development method** *Ref*: Chapter 7	Development iterations report
8	**Development iteration (1):** analyse, design, code, and test a ***micro*** prototype	Software prototype (1)

Weeks	Descriptions	Tutorials
9	**Development iteration (2):** analyse, design, code, and test a ***small*** prototype	Software prototype (2)
10	**Development iteration (3):** analyse, design, code, and test a prototype for ***sub-model 1***	Software prototype (3)
11	**Development iteration (4):** analyse, design, code, and test a prototype for ***sub-model 2***	Software prototype (4)
12	**Development iteration (5):** analyse, design, code, and test a prototype for sub-model 3	Software prototype (5)
13	**Development iteration (6):** analyse, design, code, and test a prototype for sub-model 4	Software prototype (6)
14	**Development iteration (7):** Software integration (1)	Software prototype (7)
15	**Development iteration (8):** Software integration (2)	Final software prototype

Notation

We list below the key notational styles that we use throughout the book:

- Technical concepts and Java code are written in `fixed font`. For example, class `Student` refers to the Java class named "Student".
- The singular and plural forms of a technical concept may be used to refer to instance(s) (i.e. object(s)) of this concept. For example, the phrase 'a `Student` named Le Van Anh" refers to a specific object of class `Student`, while the plural form `Students` refers to objects of the class.
- Names of software frameworks and software are written in ROMAN ALL CAPS FONT.

Acknowledgements

I would like to thank Dr. Duc-Hanh Dang and Assoc. Prof, Dr. Viet-Ha Nguyen from the Faculty of IT (VNU University of Engineering and Technology), whose guidance and wisdom in language engineering helped shape a domain modelling language from the meta-attributes used in this book. It was through the development of this language that I was able to improve the design of the meta-attributes. I wish to also thank my colleagues and students (both past and present) at the Faculty of IT, Hanoi University who have been using the jDomainApp framework and DomainAppTool in teaching and studying and contributing useful feedbacks. Both the framework and tool would most definitely not be what they are today without the many opportunities for being applied in teaching that they have had.

Chapter 1. Case study: COURSEMAN

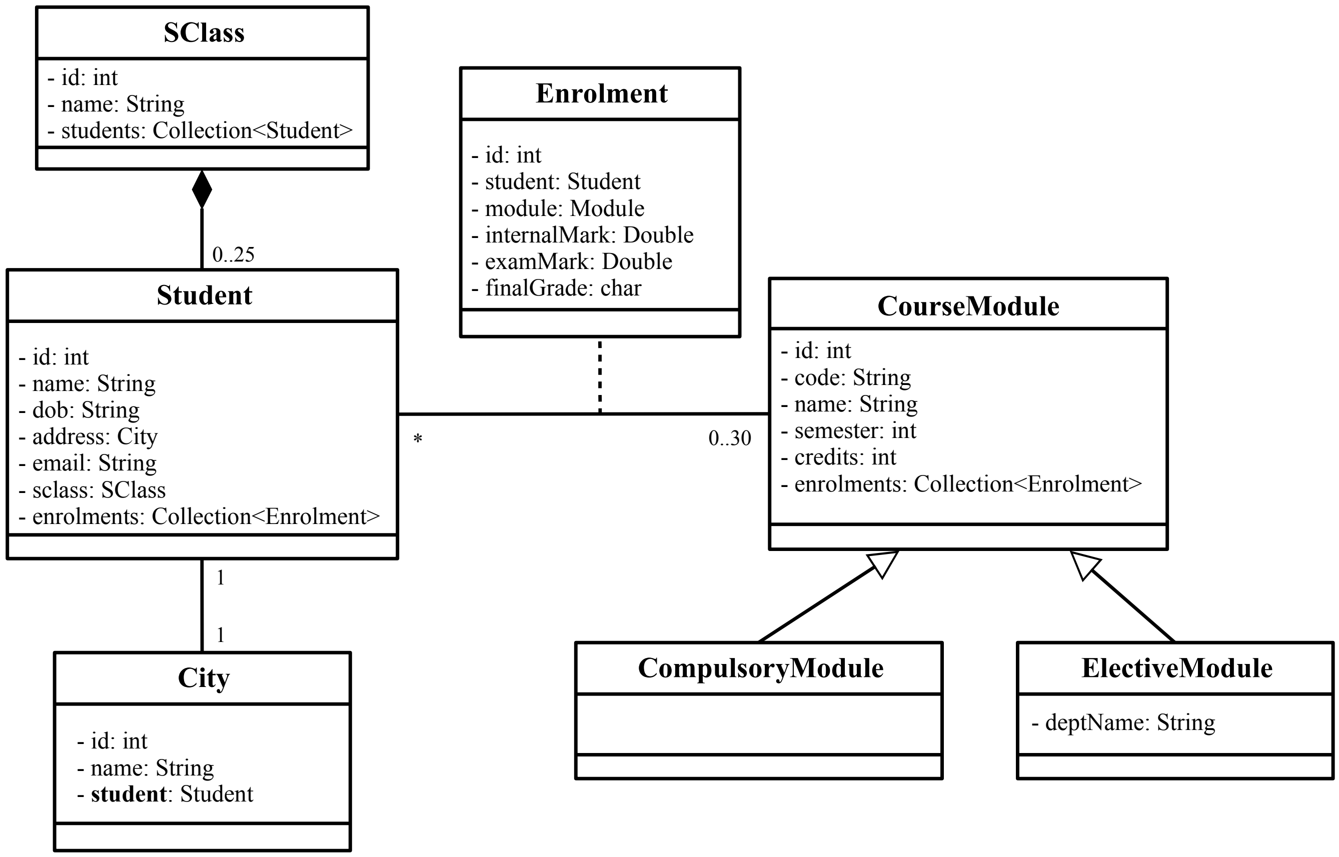


Figure 1: The core COURSEMAN domain model.

To illustrate the concepts, techniques and method presented in this book, we use a software development case study named Course Management (COURSEMAN). This software is used to manage course modules that are offered by an educational institution and the students that are registered to study in these modules. We explain in this chapter the requirements of COURSEMAN. These requirements are based on the author's own experience in using the course management system at the Faculty of IT (Hanoi University). Variants of this case study have been used as examples in several academic research papers ([16]–[20]) that are foundational for this book.

We organise the requirements according to three subsections. Subsection 1.1 describes the core domain requirements. Traditionally in DDD, these requirements explain the concepts that make up the domain model. Subsection 1.2 extends the core requirements to describe a special type of concept named *report*. Subsection 1.3 summarises the requirements of the functions that manipulate the concepts described in the other two subsections.

1.1. Domain model

Figure 1 shows the domain model of CourseMan. This model is expressed as a UML class diagram [21]. It consists of 7 domain classes: `Student`, `SClass`, `CourseModule`, `CompulsoryModule`, `ElectiveModule`, `Enrolment` and `City`. Each domain class realises a domain concept. The following paragraphs will describe these classes.

`Student` represents students who are registered to study in the institution. It is characterised by the following basic attributes: `id` (identifier), `name`, `dob` (the abbreviation for date of birth), `address` and `email`. In particular, 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` that registers to study in the year 2016 will have the `id` of S2016. The second student of that year has the `id` of 2017, and so on.

`City` provides a simple abstraction of an address. `City` represents only the city part of address and is characterised by two attributes: `id` and `name`.

Student class (class: **`SClass`**) represents a logical grouping of `Students` into administrative class unit. `SClass` is described by `id` (identifier) and `name`. An `SClass` can have no more than 25 `Students`.

Course module (class: **`CourseModule`**) represents a taught course module. It is characterised by the following attributes: `id` (identifier), `code`, `name`, `semester` and `credits`. Attribute `semester` has values in the range [1,10], while `credits` must be greater than 0. Attribute `code` is automatically generated by combining the letter 'M' with a number that is computed by multiplying the semester by 100 and adding the result to an auto-incremented counter, that starts from 0. For example, the `code` of the first module in semester 1 is the literal "M100", that of the second module is "M101", and so on.

There are two types of course modules: **compulsory** (class: **`CompulsoryModule`**) and **elective** (class: **`ElectiveModule`**). In addition to the common attributes, an `ElectiveModule` 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 that in the home faculty of the students.

`Enrolment` is an association class that realises the many-many association between `Student` and `CourseModule`. It records a fact that a `Student` has registered to study in a specific `CourseModule` in a given semester. It also holds data about the internal mark (`internalMark`), examination mark (`examMark`), and the final grade (`finalGrade`) that the student has obtained in the module. Attribute `finalGrade`, in particular, is a single character which must be one of the followings: "E" (excellent), "G" (good), "P" (pass) and "F" (failed).

Note the following cardinality constraint on the association between `Student` and `CourseModule`: a `Student` may not be registered to study in more than 30 `CourseModules`.

Associative attribute

In addition to the normal class attributes described above, each domain class in the domain model in Figure 1 contains one attribute that realises its end of an association to another domain class in the model. We call these attributes **associative attributes**. We list below the associative attribute pairs by the associations shown in the figure:

- association (`SClass`, `Student`): `SClass.students`, `Student.sclass`
- association (`Student`, `City`): `Student.address`, `City.student`
- association (`Student`, `Enrolment`): `Student.enrolments`, `Enrolment.student`
- association (`Enrolment`, `CourseModule`): `CourseModule.enrolments`, `Enrolment.module`

The **data type** of an associative attribute is either the domain class at the opposite end of the association or a generic type `Collection<T>`, whose type variable `T` is the domain class at the opposite end. The former case occurs when the cardinality of the opposite end is one, while the latter occurs when the cardinality of the opposite end is many. Let us take the association (`SClass`, `Student`) as an example. The data type of attribute `Student.sclass` is `SClass` because the opposite end of the association is `SClass` and has the cardinality of 1. On the other hand, the data type of attribute `SClass.students` is `Collection<Student>`, because the opposite end of the association is the class `Student` and has the cardinality of "many".

1.2. Reports

In addition to the domain classes described in the previous section, CourseMan's domain model also contains reports. One such report is called student-by-name, which provides information about students whose name match a user-specified name pattern. For example, if the user specifies a name pattern “`Le Van Anh`” as input then the report will show all students whose names match this name exactly. However, if the user specifies the name pattern “`Le%`” then the report will show all students whose names start with the string “`Le`”.

1.3. Functional requirements

Two primary functions of the CourseMan software are to allow a faculty to **manage students** and **manage the course modules** that are offered to the students. In addition, the software is to allow the faculty to **enrol a student** into the course modules in each semester, and to enter the module marks for each enrolment. The software needs to compute the final grade for each enrolment from its marks. Further, the software needs to allow the faculty to **manage student classes** from groups of students.

As far as reporting is concerned, the software enables the user to execute the **student-by-name report** described in the previous section.

Exercises

Ex1. Identify and briefly describe the element types of UML class diagram that are used to express the CourseMan domain model.
Note: use sections §9.2.2, 9.5,11.4 and 11.5 of the UML specification [21] as reference.

Ex2. What element type(s) are missing from the CourseMan domain model? Briefly describe them.
Note: use the relevant section(s) of the UML specification as a reference.

Ex3. Why were the element type(s) identified in Ex2 not included in the CourseMan domain model?

Ex4. For each domain class of the CourseMan domain model that has the missing element type(s), add directly to the model some missing elements of these type(s). Justify your answer.

Ex5. Discuss at least one other way of defining the data type of the collection-typed associative attribute.

Ex6. Identify and briefly describe the requirement for at least two other reports that can be defined based on the CourseMan domain model.

Chapter 2. DomainAppTool: An architectural overview

DomainAppTool is designed based on the MVC software architecture [5], [16], [22] to implement the DDD philosophy. Conceptually, the tool's architecture is built from three key components: model manager, view manager and object manager. In this chapter, we will explain these three components and, in particular, give a detailed description of the view manager's output – the graphical user interface (GUI) of the software.

2.1. An overview of DDD and DomainAppTool

In 2004, Evans [7] formulated a domain-driven design (DDD) method for developing the domain model. DDD offers a domain-specific outlook for software development methods like OOD. Although DDD is not exclusively for OOD, the two are a natural match. Conceptually, a *domain* is a specific area of interest to software development. In DDD, the domain requirements are analysed to construct a **domain model**. DDD considers this model to be the "heart" of software, because it is where the complexity lies. With this view, the DDD method gathers the best design practices for frequently-occurring domain modelling problems and defines them as patterns – "tools" that designers can use to solve the design problems for their own software.

The DDD method itself does not provide any tools for software developers to use. This leads to independent groups proposing tools for the method. Among the available DDD tools are Apache-IsIs [8], [9] (the successor of Naked Objects [10]) and OpenXava [11], [12]. The former is perhaps a model DDD tool, to which Evans has been contributing. The latter is a mature tool for developing real-world software projects.

In 2012, a foundation was laid by this book's author with the development of a domain-driven software framework named jDomainApp [13]. This framework was originally conceived to be used only as a tool for teaching an object-oriented program development method in the Java programming language [14], [15]. This method, which is proposed in an authoritative book [4], was updated and adapted by this book's author to incorporate the DDD philosophy. The improved method was then used to teach two core software engineering course modules at the Faculty of IT (Hanoi University).

In 2014, the development of DomainAppTool, which is based on jDomainApp, began when a need was realised for a user-friendly tool that can be used to quickly and interactively execute and test the domain model. This tool has been tried at the Faculty of IT in teaching another software engineering course module and proved useful.

2.2. Model manager

This component of DomainAppTool is responsible for managing the domain model. A domain model consists of a set of domain classes and the associations between them. A domain class is a Java class that is designed with domain-driven design features. Each class captures the domain requirements of a concept or entity of interest. Chapter 3 explains how to develop a domain class in DomainAppTool.

A key feature of the tool is that it only requires the developer to specify the domain model (in the form of a set of domain classes) of a software. The entire software (which includes a graphical user interface (GUI) and an object storage) is generated automatically at run-time from the domain model. Chapter 5 will explain how to run the tool with the model.

2.3. View manager

A software that is generated by the tool has a GUI, which is used by the user to perform the software functions. A feature of the tool is that the software GUI is automatically generated at run-time from the design information embedded in the domain model. **View manager** is the component responsible for this task. In principle, the view manager provides a **desktop manager** for managing the different object forms. An object form consists of one or more Swing-based components, each of which presents the view of a domain attribute. An **object form** is a key part of an internal window that provides an interface for the user to view and to manipulate the domain objects of a domain class. Section 2.5 will give more details about the view's structure and how it is used to interactively manipulate the domain objects.

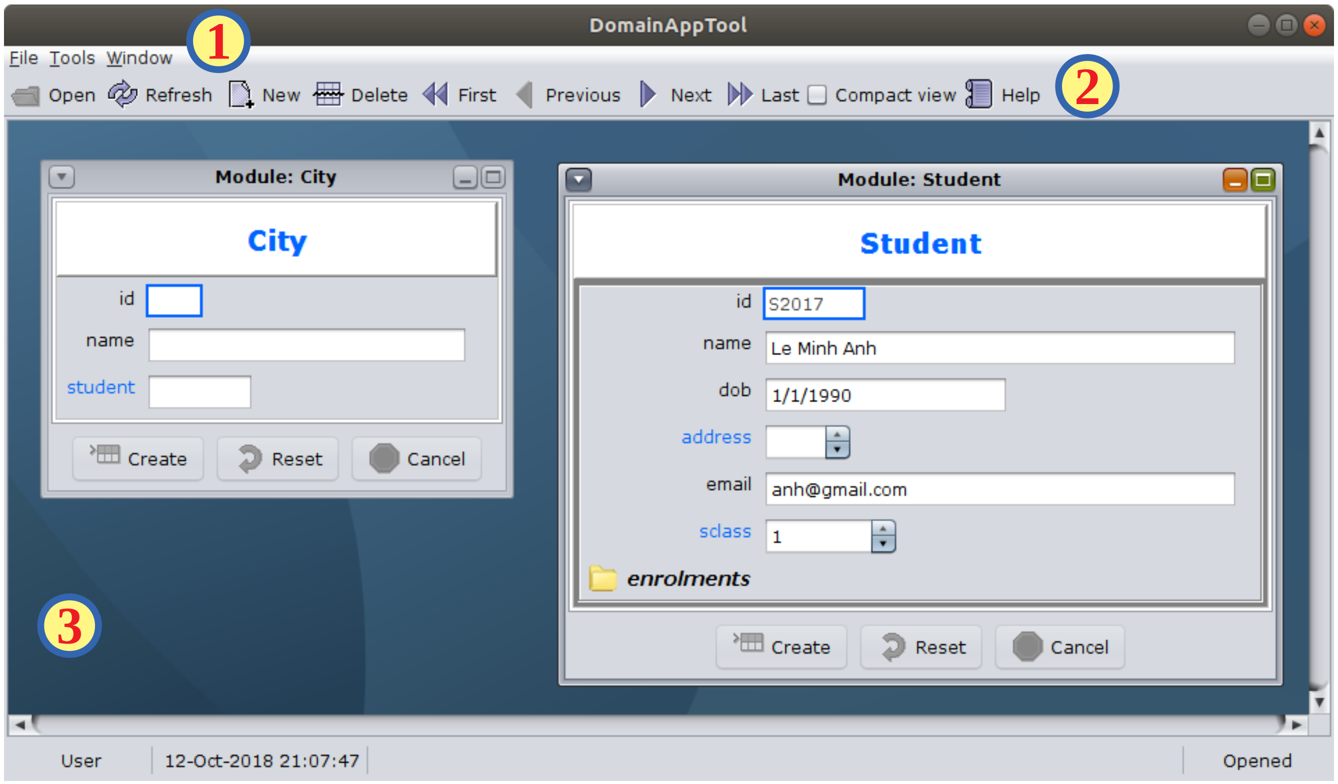
2.4. Object manager

The domain objects of a software are managed by a component called **object manager**. This component effectively manages the objects in the shared (heap) memory and provides a mechanism to store the objects into an external storage space.

For study purpose in this book, the tool supports a relational database management system called Java DB [27], [28]. This database is provided as part of the Java platform. A database is automatically created for a software the first time it is run. The database schema is created from the design information embedded in the software's domain classes. At run-time, the domain objects of the software are converted into table rows and stored in this database. We will discuss the software's database in more detail in Chapter 5.

2.5. Working with the software GUI

In this section, we will briefly explain how to use the software GUI generated by DomainAppTool.



id: S2017
name: Le Minh Anh
dob: 1/1/1990
email: anh@gmail.com
sclass: 1
12-Oct-2018 21:07:47

Figure 2: The main and functional views of the CourseMan software.

2.5.1. GUI basics

The software GUI is made up of two types of container: main window and internal window. The main window, which is technically named the **main view**, is the outer window shown in Figure 2. It is the single top-level container that consists of a **menu bar** (labelled 1 in the figure), a **tool bar** (labelled 2), and a **desktop** (labelled 3). The menu bar is rather self-explanatory. Its Tools menu, in particular, lists the actions that invoke the software modules. Selecting a Tools's menu item will activate a module and displays its view. A **software module** is generated from one domain class for the purpose of managing domain objects of this class. We will explain the module's view shortly below.

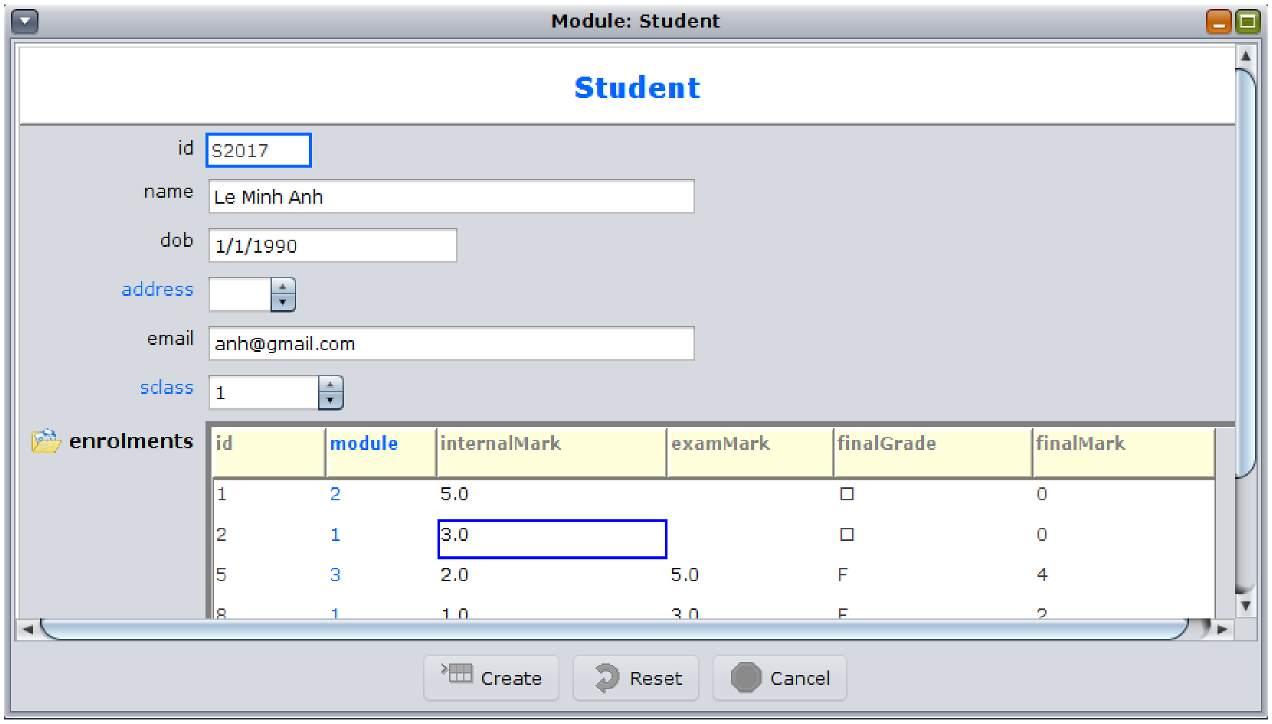
Table 2: The core tool bar and menu buttons

Tool bar and menu buttons		Description
Tool bar	Open	Connects to the data store of the domain class of the active object form, and prepares to store and retrieve objects.
	New	Creates a new object of the domain class of the active object form. It also invokes action `Open` to initialise the data store connection.
	Update	Updates the currently edited domain object (which is being presented on the active object form).
	Refresh	Refreshes the current object (presented on the active form)
	Delete	Deletes the current object. This also deletes links to other objects and also objects that depend on this object.
	First	Browses to the *first* object in the data store of the domain class of the active object form.
	Previous	Browse to the *previous* object in the data store of the domain class of the active object form.
	Next	Browse to the *next* object in the data store of the domain class of the active object form.
	Last	Browses to the *last* object in the data store of the domain class of the active object form.
	☐ Compact view	Toggles compact or normal view of the active object form. The compact view excludes the action panel from the form.
Tools menu	Find...	Turns on the search tool bar on active object form. User uses this tool bar to search for domain objects.

The tool bar contains a series of labelled buttons that are mapped to the key object manipulating actions of the software. The actions, together with their associated buttons, are summarised in Table 2. The object manipulation actions will be explained in the subsequent sections.

The desktop of the main view is a container that holds one or more internal windows. An internal window, which is technically named **functional view**, is the view of a software module. For example, the desktop in Figure 2 shows the views of two modules for `City` and `Student`. Each view is in turn a container consisting of four components: (1) **title**, (2) **object**

form, (3) **action panel**, and (4) **search tool bar**. We will briefly explain the first three components here. We will discuss the search toolbar later in Section 2.5.7. Note that this tool bar is not visible by default and its visibility can be turned on and off by the “Find” menu item of the Tools menu.


Module: Student

Student

id	S2017
name	Le Minh Anh
dob	1/1/1990
address	
email	anh@gmail.com
sclass	1

enrolments

id	module	internalMark	examMark	finalGrade	finalMark
1	2	5.0		☐	0
2	1	3.0		☐	0
5	3	2.0	5.0	F	4
8	1	1.0	3.0	F	2

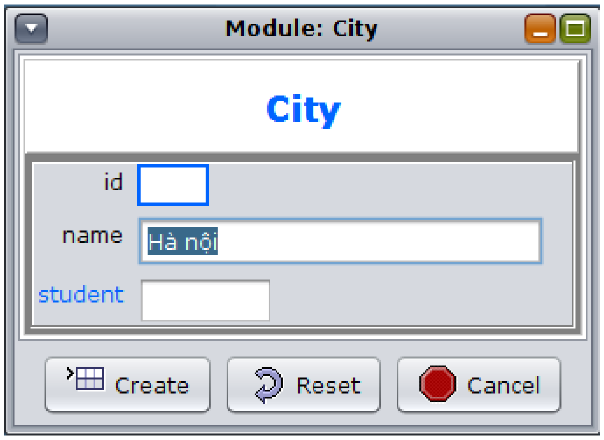
Figure 3: The sub-form of associative attribute `Student.enrolments`.

A view’s title displays the module’s title, which in the examples shown in Figure 2 is set to the name of the domain class. The object form is a container for the data components that render the domain attribute values of an object. There are two types of data components: **data field** and **sub-form**. The former is used for both non-associative attribute (e.g. `Student.name`) and non-collection-typed associative attribute (e.g. `Student.address`). The latter is primarily used for collection-typed associative attributes (e.g. `Student.enrolments`). The action panel contains three action buttons that are labelled `Create`, `Reset` and `Cancel`. These buttons represent three common input actions that are performed with regards to the data that are displayed on the object form. Action `Create` will perform object creation from the input data. Action `Reset` will reset the object form to its initial state. Action `Cancel` will cancel the object creation operation and set the object form to its initial state. More details on how to perform these actions will be given in the subsequent sections.

The object form of a domain class that has collection-typed associative attributes will contain sub-forms that present views of the associated domain classes. A sub-form is labelled from the name its corresponding associative attribute. By default, the sub-forms are collapsed and the user is required to click on the label to expand it into view. For example, the `Student`'s object form shown in Figure 2 has a sub-form for the associative attribute `Student.enrolments`. This sub-form is labelled "`enrolments`" and is displayed at the bottom of the `Student`'s form. Figure 3 shows this sub-form after it has been expanded.

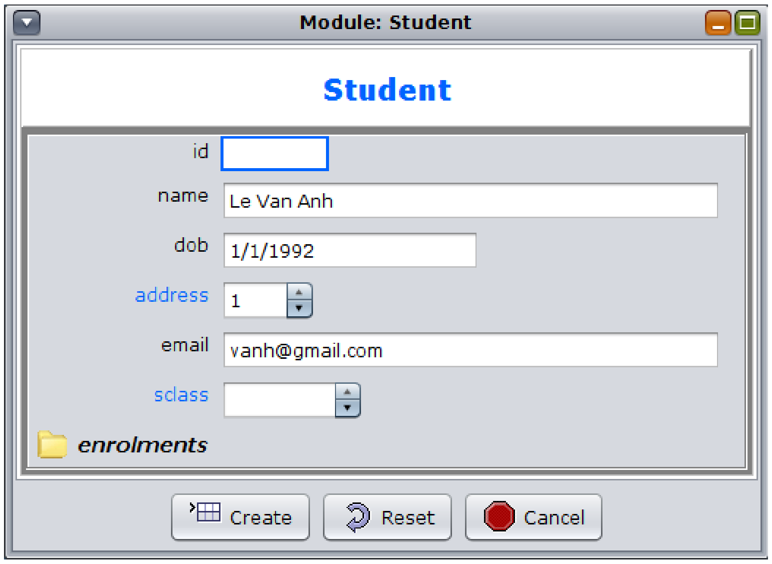
2.5.2. Create a new object

The first object manipulating function that can be performed via the GUI. The behaviour is to create a new object of a domain class. The associated tool bar button is the button `New`. This button is enabled for each module whose domain class is not abstract and when the module's view is selected in the desktop.



id	
name	Hà nội
student	

Figure 4: Creating a new `City`.



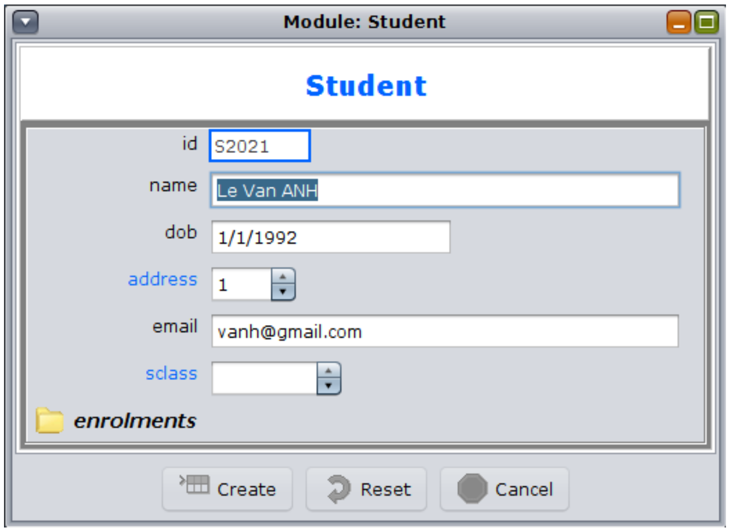
id	
name	Le Van Anh
dob	1/1/1992
address	1
email	vanh@gmail.com
sclass	

Figure 5: Creating a new `Student`.

When the user clicks on this button, the selected view will be cleared with its three action buttons enabled. The user then enters data into the data components of the object form (as exemplified in Figures 4 and 5 for `City` and `Student`, respectively) and uses one of the three action buttons to act on these data.

2.5.3. Update an object

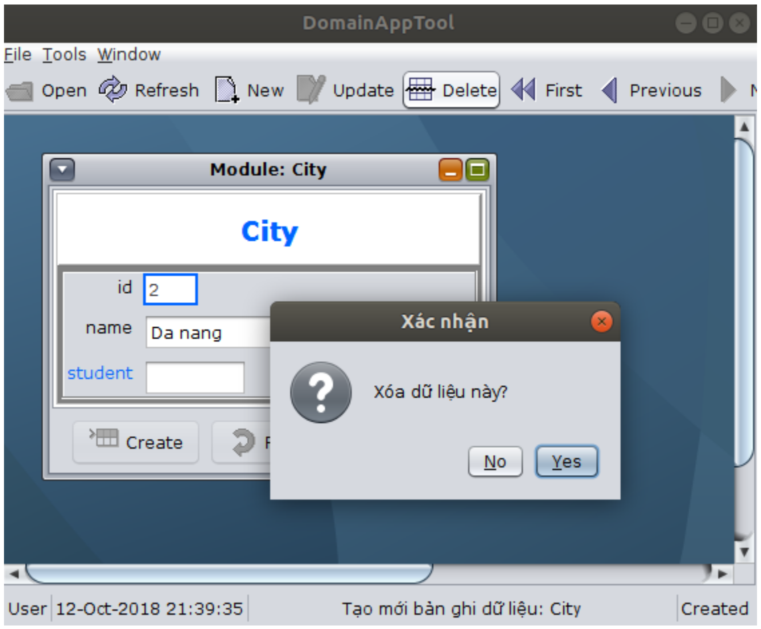
When an object state needs to be changed, the user will use the object form to input the changes. Figure 6 gives an example of how the user uses the `Student`'s object form to enter a new value for the attribute `name`. To reverse the changes, the user uses the action button `Reset` (which is enabled in the figure). To commit the changes, however, the user needs to press the tool bar button `Update` (which is also enabled in the figure).



id: S2021
name: Le Van ANH
dob: 1/1/1992
address: 1
email: vanh@gmail.com
sclass:
enrolments

Figure 6: Updating a `Student` by changing her `name`.

2.5.4. Delete an object



id: 2
name: Da nang
student:

Xóa dữ liệu này?

User | 12-Oct-2018 21:39:35 | Tạo mới bản ghi dữ liệu: City | Created

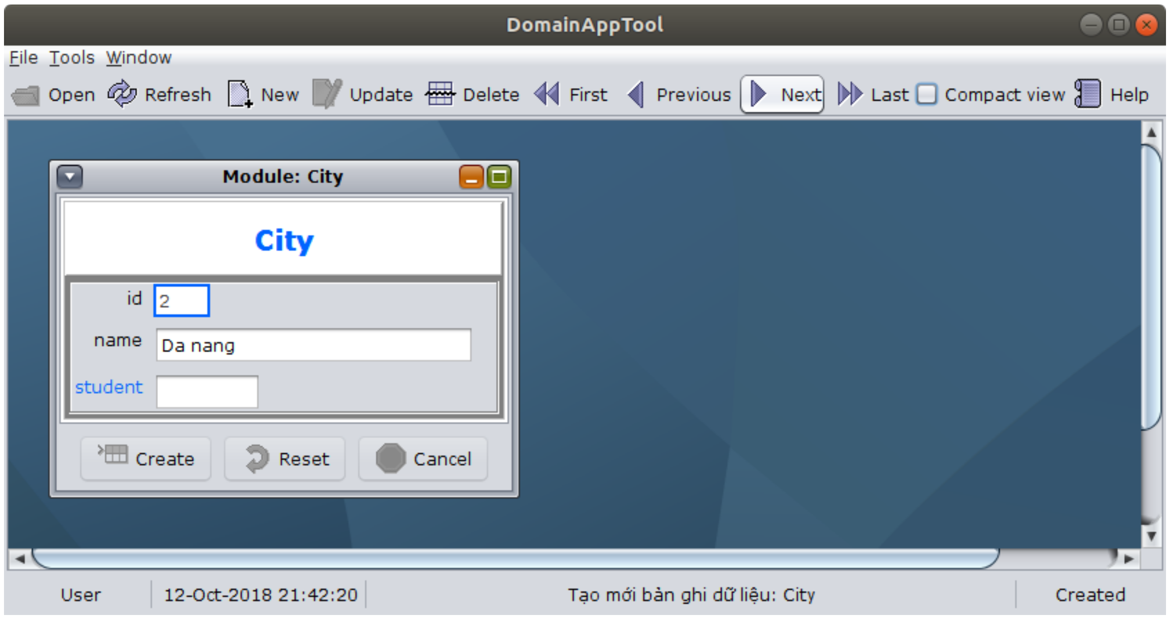
Figure 7: Deleting a `City`.

To delete an object, the user first ensures that the object is being active on the object form. She then presses the tool bar button `Delete` and answers “Yes” on the subsequent confirmation dialogue. Figure 7 illustrates this with the `City` object named “Da nang”.

2.5.5. Browse objects

When the software indicates to the user that there exists a set of objects of interest and the user would like to view each one of them in turn on the object form, she will need to use the four browsing buttons that are located towards the end of the tool bar. These buttons are intuitively labelled `First`, `Previous`, `Next` and `Last`. They are enabled or disabled depending on the location of the active object (if any) in the browsing sequence. By default, this sequence is defined based on the natural ordering of the id attribute's values.

Figure 8 illustrates how the browsing buttons are used to browse through all the `City` objects in the ascending order of the id attribute's values.



id: 2
name: Da nang
student:

User | 12-Oct-2018 21:42:20 | Tạo mới bản ghi dữ liệu: City | Created

Figure 8: Browsing `City` objects

2.5.6. Navigating through object links

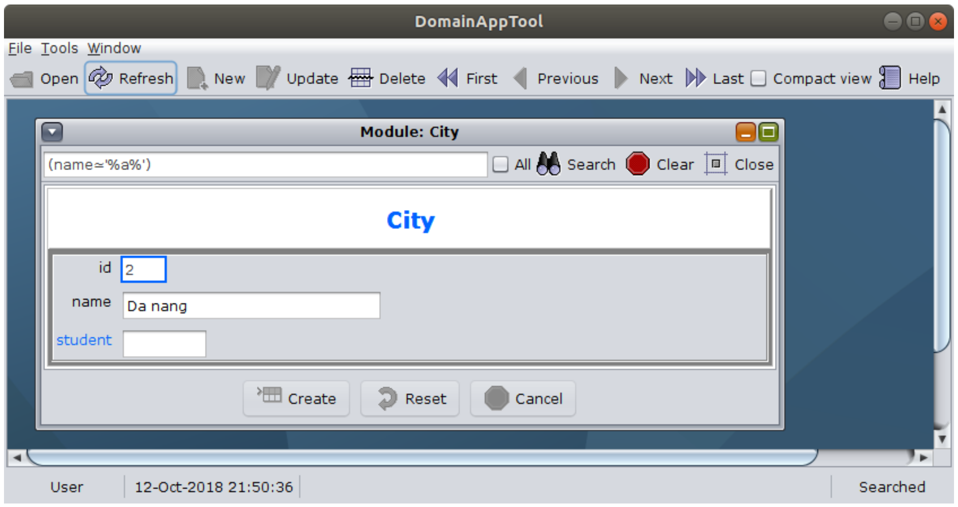
An object that has association links to other objects (via the associative attributes) will be presented with links that can be used to navigate to those objects. The links are attached to the labels of the associative attributes. These labels are coloured in blue which invite the user to click on them. When this occurs, the object form of the associated domain class will be displayed (in the compact form), presenting the object that is currently participating in the link. If this object has links to other objects then the user can navigate these links to display those objects and so on.

For example, the `Student`'s object forms shown in Figure 3 and Figure 5 both have links attached to two associative attributes `Student.sclass` and `Student.address`. However, the link for Student.sclass is only active in Figure 3 because it has an object value set on the attribute. Similarly, the link for Student.address is only active in Figure 5.

Note that to activate a link on a tabular form, the user is required to press and hold Ctrl and move the mouse over to and click on the target cell on the column corresponding to the attribute. For example, to activate the link to the CourseModule object with id = 2, which is attached to attribute Enrolment.module shown in the Enrolment's sub-form in Figure 3, the user is required press and hold Ctrl, move the mouse over to the cell containing the value 2 of the column named "module" and then click on this cell.

2.5.7. Search for objects

While the user can browse through all the objects as explained above, it is more common, however, for the user to be interested in only a sub-set of the objects – especially those whose states match a pre-defined search pattern. In this case, the user will first need to search for the object sub-set before using the browsing buttons to browse them. Figure 9 shows an example of how to achieve this with City objects.



(name≃'%a%')

id: 2

name: Da nang

student:

User | 12-Oct-2018 21:50:36

Figure 9: Searching for City objects whose names contain the search term "a".

In this example, the user is interested in viewing the subset of City objects whose names contain the character "a". To search for these objects, the user first needs to open the search tool bar of the object form, by pressing the Tools menu item labelled Find while the object form is being selected. Next, the user enters search values for each attribute of interest (similar to when she input data to create or update the object). In the example in the figure, for instance, the user enters the search value "`a`" on the `name` data field. After that, the user adds the search values to the search query by pressing the `Ctrl` key while left-clicking on the labels of the corresponding data fields. The search query content is automatically updated and displayed in the left-most text box of the search tool bar. When the search query is updated, the button `Find` of the search tool bar is enabled. When the user clicks on this button, the search query is executed to find the matching objects. If such objects are found then they are presented on the object form for the user to browse (as explained in Section 2.5.5).

Exercises

Ex1. What is the MVC architecture model? Illustrate your answer with an example.

Ex2. What constitutes a domain model? Why do we need to manage the domain model (using the model manager)?

Ex3. How do we construct a software using the DomainAppTool?

Ex4. What does the software GUI consist of? What are the main software function(s) that can be performed using this GUI?

Ex5. How are the domain objects stored in DomainAppTool?

Ex6. Apply the GUI instructions concerning the object manipulation actions given in this chapter to the sub-form of the associative attribute `Student.enrolments`. Observe and record the *differences* between performing an action on this sub-form and performing it on the `Student`'s form.

Ex7. Could you perform *all* actions on the sub-form of the attribute `Student.enrolments`?

Ex8. Use the object forms of `SClass` and `Student` to experiment with two methods of creating a new `Student`. What is the main difference between these two methods?

Ex9. Use the `Student`'s form to delete a `Student` object. Then use the `Enrolment`'s form to search for the `Enrolments` that were associated to the deleted `Student`. Are they there? Explain.

Ex10. Use the `SClass`'s object form to delete a `SClass` object. Then use the `Student`'s form to search for the `Students` that were associated to the deleted `SClass`. Are they there? Explain.

Ex11. Use the object forms of `Student` and `City` to observe and explain what happens when you delete a `City` object.

Chapter 3. Developing the domain model

In the spirit of the DDD method, DomainAppTool considers the domain model as the heart of software. This section explains how to develop the domain model for DomainAppTool. We first give an overview of the design method. After that, we explain a set of annotations (called *meta-attributes*) that we use to design the domain classes. These include annotations that are attached not only to the domain attributes but to the operations.

3.1. Design overview

A **domain class** is a class that is designed with domain-specific information. According to [29], 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 and to its members. A **meta-attribute**[3] consists in a set of properties, whose values are specified when the attribute is attached to a target element.

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

We adapted the design approach in [29], which models a meta-attribute as a class. The attachment of this class to a target elements creates an annotation element (an object of the annotation). The annotation element and its attachment to the target element is drawn using the UML's note box. The annotation element's state is specified using this textual form: `@A { props }`, where A is the annotation's name, `props` is a list of property initialisations. This initialisation has the form `p = v`, where `p` is a property's name and `v` is a value. The examples given shortly below will illustrate this notation.

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

3.2. Meta-attribute `DClass`

This meta-attribute has four basic properties: `schema`, `serialisable`, `singleton` and `mutable`. Property **`schema`** specifies the name of the storage schema, in which the domain

3 the term 'meta-attribute' that we use here is synonymous to 'attribute' in [29].

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 are stored at the storage space whose name is specified by the property `schema`.

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. In particular, if the property is `false` 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 of the meta-attribute `DAttr` (introduced shortly below) that has the same name.

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

Example: CourseMan

Figure 10 shows how meta-attribute `DClass` is modelled in the UML class diagram of the CourseMan's domain model. The complete source code of the CourseMan domain model is provided in the file `courseman-source.zip`, which is attached to this book. The byte codes are included in the package named `vn.com.courseman.model` in the example library file named `courseman.jar`.

The default values of the four `DClass`'s properties are listed in the specification of the meta-attribute for the class `SClass`. The default values of `schema` and `serialisable` mean that all domain classes of the CourseMan software are serialisable to the schema named `App`. On the other hand, the default values of `mutable` and `singleton` mean that all domain classes of the CourseMan software are mutable and can have any number of domain objects.

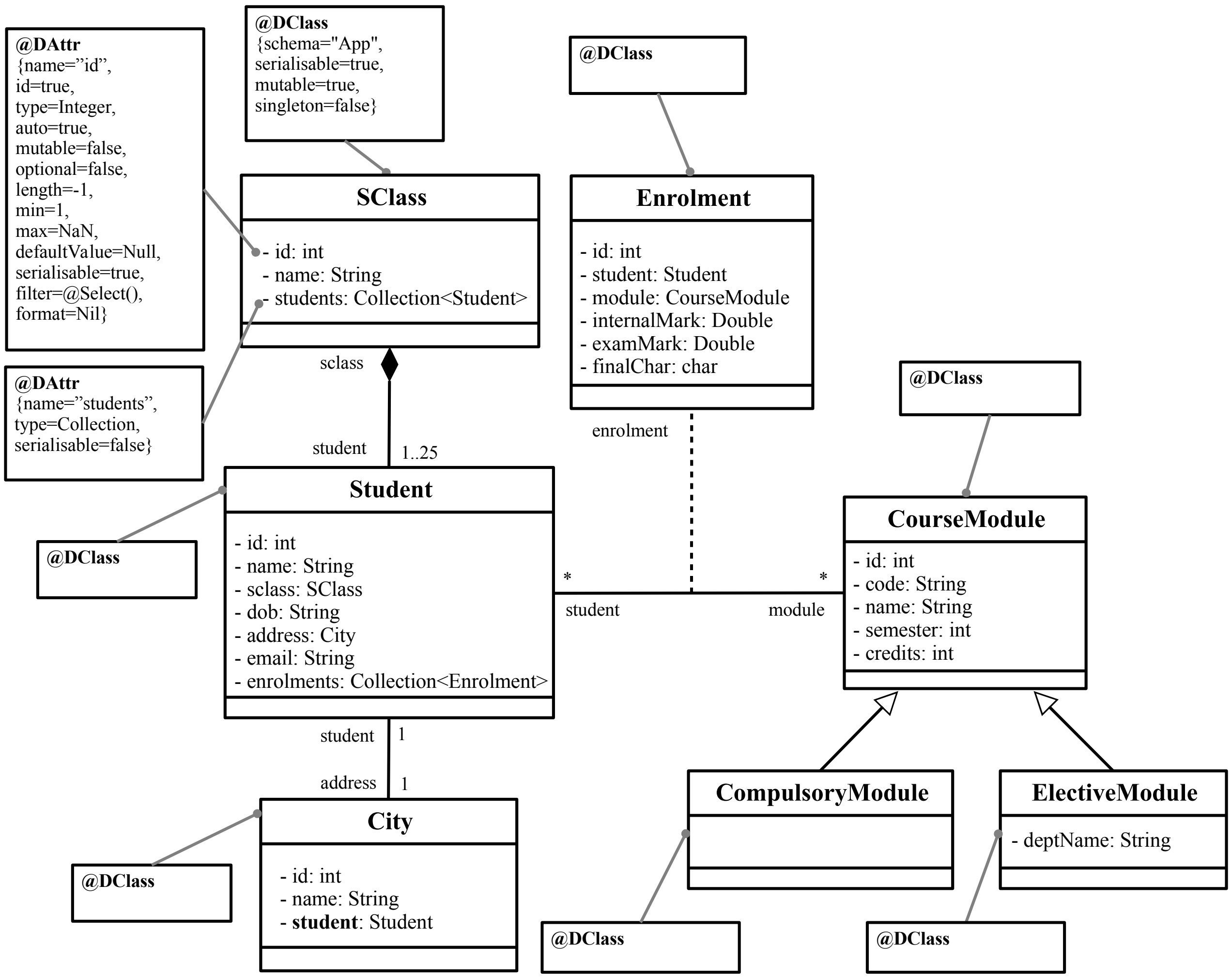


Figure 10: The basic design of the domain classes in the COURSEMAN domain model.

3.3. Meta-attribute `DAttr`

Throughout this book, 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 [7]. This set has nine properties: **`name`**, **`type`**, **`mutable`**, **`optional`**, **`length`**, **`min`**, **`max`**, **`defaultValue`** and **`format`**. These properties are rather self-explanatory, whose detailed information can be found in [30][4]. The second set has four 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

4 Students should additionally refer to the prerequisite OOPL knowledge.

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` (which is the default value) 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 one-many associations with other classes. The value of this property is another annotation named `Select`. This annotation has two properties: `clazz` and `attributes`, which together specify a sub-set of the domain attributes of the specified domain class, whose values are of interest in the collection. We will explain collection-typed attributes shortly below and discuss association in Section 3.4.

Example: CourseMan

Let us illustrate `DAttr` using two domain attributes `SClass.id`, `students` of the CourseMan model shown in Figure 10. 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 10 are given in Appendix 1.

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 the default and 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 one-many association with another class. In this example, the owner domain class of the attribute is `SClass` and the associated class is `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. The value of property `filter` does not specify any specific domain attributes of `Student`, which means that all `Student` attributes are of interest to the collection.

Listing 1: Java specification of the domain attributes of `SClass`

```java
import java.util.ArrayList;
import java.util.Collection;

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 Collection<Student> students;
}
```

3.4. Meta-attribute `DAssoc`

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

`DAssoc` consists in the following properties that describe both association and association end: `ascName`, `role`, `ascType`, `endType`, `associate`, `dependsOn`. Property **`ascName`** specifies the name of the association. It uniquely identifies an association and is therefore used to determine the set of `DAssoc`s 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 `DAssoc`s already in the set. By convention, the value of property `ascName` 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 between two classes

Property **`role`** specifies the association's role of the domain class that owns 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` and `many-to-many`. To ease maintenance, these association types are pre-defined in the enum `DAssoc.AssocType`. Similar to property `ascName`, the value of this property needs also be the same for the `DAssoc`s in the same set.

Property **`endType`** specifies the type of association's end, which is either `one` or `many`. These end types are pre-defined in the enum `DAssoc.AssocEndType`. 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` and `determinant`. Property **`type`** specifies the associated domain class of the opposite end. The value of this property is written using the Java's class notation, which consists 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. 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 [30]). The value of this property is `true` only for 1-1 associations whose one end is a strong entity.

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 is applied to any type of association (not just one-to-one).

Example: COURSEMAN

Figure 11 illustrates how `DAssoc` is used to define three associations of the COURSEMAN domain model. For brevity, we only show the `DAssocs` of the following six associative attributes that realise the associations: `SClass.students`, `Student.sclass`, `Student.address`, `City.student`, `Student.enrolments` and `Enrolment.student`.

As shown in the figure, for instance, the `ascName` and `ascType` of the two `DAssocs` of the association between `SClass` and `Student` are both set to "sclass-has-students" and "one-many" (respectively). The `role` and `endType` of the `DAssoc` attached to `SClass.students` are "sclass" and "one" (resp.), while those of the `DAssoc` attached to `Student.sclass` are "student" and "many" (resp.) The `associate` of the `DAssoc` at one end is defined based on the opposite end of the association. For example, the `associate` of `DAssoc`(`SClass.student`) has its `type` set to `Student.class`, and its `cardMin` and `cardMax` set to the min and max cardinalities of `Student`.

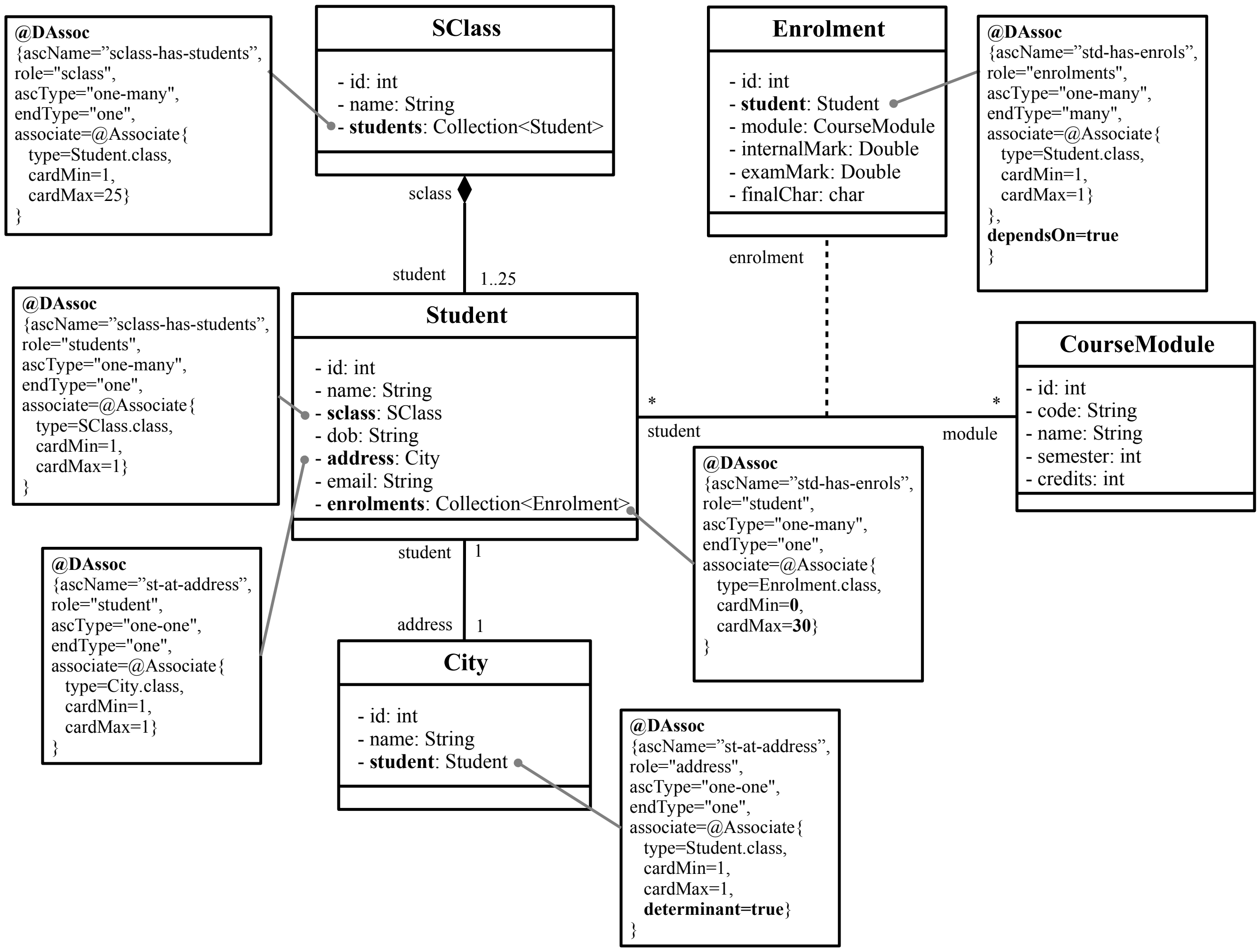


Figure 11: Applying `DAssoc` to (partially) define three associations of CouseMan.

The values of properties `cardMin` and `cardMax` in the figure are self-explanatory, except for those defined for the associative 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 `ascType` "one-many" of the association.

Property `determinant` shown in the figure means that `Student` is the determinant of the association between it and `City`. Property `dependsOn` means that `Enrolment` depends on `Student`.

Listings 2 and 3 below show the Java code snippets of two domain classes `City` and `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`.

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

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

```java
@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;
}
```

3.5. Operations

Once the foundational structure of a domain class has been constructed using the meta-attributes, the designer then proceed to defining 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 [4][5]: 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 Appendix 1.

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

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

  // derived attributes
  private int studentsCount;

  // create objects directly from code
  @DOpt(type=DOpt.Type.ObjectFormConstructor)
  @DOpt(type=DOpt.Type.RequiredConstructor)
  public SClass(@AttrRef("name") String name) {
    this(null, name);
  }
```

5 Students should refer to the prerequisite OOPL knowledge.

```
// create objects from data source
@DOpt(type=DOpt.Type.DataSourceConstructor)
private SClass(@AttrRef("id") Integer id, @AttrRef("name") 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;
  }
}

@DOpt(type=DOpt.Type.Getter)
public int getId() {
  return id;
}

@DOpt(type=DOpt.Type.Setter)
public void setName(String name) {
  this.name = name;
}

@DOpt(type=DOpt.Type.Getter)
public String getName() {
  return name;
}
```

```
@DOpt(type=DOpt.Type.Setter)
public void setStudents(Collection<Student> students) {
  this.students = students;
}

@DOpt(type=DOpt.Type.Getter)
public Collection<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(Collection<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(Collection<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 shows two constructors, setter and getter operations for the attributes and five operations related to the associative attribute `students`. These five operations concern adding and removing `Student` object(s) that are associated to an `SClass`. All operations are specified with a helper meta-attribute named `DOpt`.

The type of a parameter used in each constructor must be an object-type (which includes Java built-in wrapper classes). The two-argument constructor invokes a class (i.e. `static`) operation named `nextID`, which produces 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. Section 3.6.2 will explain about 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`. 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: link-adder, link-remover, and link-updater. In general, these operations are used to maintain values of the associative attributes.

3.5.1. Link-adder operation

This is one of the three types of operation that are specifically designed for maintaining the value of associative attribute. 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 shared memory 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 the operation `addNewStudent` of the same listing. These 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-new operation 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 will be discussed later in Section 3.7.4.

Listing 5: Java code snippets of `City` showing a link-adder-new operation for its one-one association with `Student`

```java
@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;
  }
}
```

The link-adder-new operation in Listing 4 applies specifically to one-many associations. For one-one associations, the specification of this operation is much simpler and involves naming the operation with the prefix `setNew`. An example of this is given in Listing 5, which shows the link-adder-operation for the class `City`, named `setNewStudent`. This operation maintains the one-one 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, it would include extra statements needed to update the state of the current `City` object.

3.5.2. Link-remover operation

This is the second type of operation that is applied to associative attributes. It 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.

3.5.3. Link-updater operation

Listing 6: Java code snippets of `Student`, showing a link-updater operation named `updateEnrolment` for updating attribute `Student.averageMark`

```
@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 below shows an example of a third type of operation that is applied to associative 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`).

Section 3.6.4 will provide more detail about operation `updateEnrolment` and, more generally, about the design issue concerning the maintenance of attribute `Enrolment.finalMark`.

3.6. Design issues

We discuss in this section a number design issues concerning the following three types of domain class members: constructor, auto-generated attribute and derived attribute. Constructor operation is key to the creation of domain objects, while the two attribute types frequently appear in real-world domain models.

3.6.1. Constructor rules

In addition to the standard design rules, the constructor operation must follow a number design rules required by DomainAppTool.

Base constructor rules

CR1. *A domain class must define an **object form constructor** (i.e.* `DOpt.type=ObjectFormConstructor`*) if it has at least one non-auto and non-collection-type attribute(s).*

CR2. *A domain class must define a **required constructor** (i.e.* `DOpt.type=RequiredConstructor`*) if it has at least one mandatory attribute(s).*

CR3. *A domain class must define a **data source constructor** (i.e.* `DOpt.type=DataSourceConstructor`*) if it has at least one serialisable attribute(s).*

The constructors mentioned in the above three rules may or may not be distinct. That is, one constructor operation may fit the criteria of two or even all three rules. The object form constructor mentioned in rule CR1 is used to assign values to the user-specified attributes. The values are typically entered by the user on an object form. The required constructor mentioned in rule CR2 is used to assign values to the mandatory attribute(s) of the class. This constructor is used by the object manager to create new objects from the input data provided by the user. The data source constructor mentioned in rule CR3 is used to assign values to the serialisable attribute(s). This constructor is used by the object manager to reconstruct the domain objects from the data source.

Listing 7 shows examples of the three types of constructor that are defined in the domain class `Student`. Note how the first constructor is annotated as both the object form and the required constructor.

Listing 7: Three constructors of `Student` with suitable `DOpt.type` and `AttrRef`

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

  @DOpt(type=DOpt.Type.ObjectFormConstructor)
  @DOpt(type=DOpt.Type.RequiredConstructor)
  public Student(@AttrRef("name") String name,
      @AttrRef("dob") String dob,
      @AttrRef("address") City address,
      @AttrRef("email") String email) {
    this(null, name, dob, address, email, null);
  }

  @DOpt(type=DOpt.Type.ObjectFormConstructor)
  public Student(@AttrRef("name") String name,
      @AttrRef("dob") String dob,
      @AttrRef("address") City address,
      @AttrRef("email") String email,
      @AttrRef("sclass") SClass sclass) {
    this(null, name, dob, address, email, sclass);
  }

  @DOpt(type=DOpt.Type.DataSourceConstructor)
  public Student(@AttrRef("id") String id, @AttrRef("name") String name,
      @AttrRef("dob") String dob,
      @AttrRef("address") City address,
      @AttrRef("email") String email,
      @AttrRef("sclass") SClass sclass) throws ConstraintViolationException {
    // generate an id
    this.id = nextID(id);
    // assign other values
    this.name = name;
    this.dob = dob;
    this.address = address;
    this.email = email;
    this.sclass = sclass;
    this.enrolments = new ArrayList();
  }
}
```

Parameter type rule

CR4. The data types of all the parameters must be object-type.

This rule is needed to correctly handle `null` parameter values for optional attributes. Because user needs not enter values for these attributes, their values may be passed into the constructor as `null`. When this occurs and the parameter types are object types, the `null` values can be passed in successfully.

Parameter list rule

CR5. The parameters in the parameter list of a constructor must be specified with suitable `AttrRef`s*, whose values are set to the names of the corresponding attributes of the owner domain class.*

CR6. The parameters of a data source constructor must appear in the same order as they are declared in the class.

For example, all the constructors of the domain class `Student` that are shown in Listing 7 have their parameters defined with `AttrRefs`, which point to the corresponding attributes of this class.

3.6.2. Maintaining an auto-generated attribute

An auto-generated (abbreviated auto) attribute is an attribute that is specified with `DAttr.auto = true`. The previous examples show two types of auto attributes that are often found in a domain class: identifier and derived attribute. Derived attribute will be discussed in more detail in Subsection 3.6.3.

Listing 8 Maintaining the value of the auto attribute `SClass.id`

```
@DClass(schema="courseman")
public class SClass {
  /**
   * @requires
   *  minVal != null /\ maxVal != null
   * @effects
   *  update the auto-generated value of attribute <tt>attrib</tt>,
   *  specified for <tt>derivingValue</tt>, using <tt>minVal, maxVal</tt>
   */
  @DOpt(type=DOpt.Type.AutoAttributeValueSynchroniser)
  public static void updateAutoGeneratedValue(
      DAttr attrib,
      Tuple derivingValue,
      Object minVal,
      Object maxVal) throws ConstraintViolationException {
    if (minVal != null && maxVal != null) {
      if (attrib.name().equals("id")) {
        int maxIdVal = (Integer) maxVal;
        if (maxIdVal > idCounter)
          idCounter = maxIdVal;
      }
    }
  }
}
```

Similar to a normal attribute, an auto attribute may or may not be serialised. A design problem associated with serialisable, auto attributes is how to maintain the **base value** (that is used to compute a next attribute value), such that it is preserved between different runs of the software. For example, suppose the first run of the CourseMan software creates five `SClass` objects, the values of whose `SClass.id` attributes are from 1 to 5. The base value of this attribute is recorded by the class attribute `SClass.idCounter` = 5. To ease discussion we will refer to this attribute as the **base attribute**. In the second run of the software, the value of this base attribute is lost because it is not stored with the objects. This means that if the user creates another `SClass` object, this attribute will have its `SClass.id` attribute set to 1, which is already assigned to a previously created `SClass` object.

To resolve the above problem, DomainAppTool requires domain classes that have serialisable, auto attributes be defined with domain-specific rules necessary to update the base values for each of them. Further, these rules must be implemented in an operation, named `updateAutoGeneratedValue`, whose complete header specification is given in Listing 8. The code of this operation in the listing is for maintaining the base value of the identifier attribute `SClass.id`. It applies the integral, auto-incremented rule of the base attribute `SClass.idCounter` to determine when to set it to the casted value of `maxVal` argument.

Listing 9 shows another example of the `updateAutoGeneratedValue` operation, which is used in the domain class `CourseModule` to maintain two auto attributes. Because only one auto attribute is updated at a time, the code includes a branching `if` statement to identify which attribute is to be updated. If the updated attribute is `CourseModule.id` then the same integral, auto-incremented rule as the one used above for `SClass.id` is applied to update the base attribute `CourseModule.idCounter`. On the other hand, if the updated attribute is `CourseModule.code` then a relatively more complex auto-incremented rule is applied to update the base attribute `CourseModule.currNums`. The definition of this attribute together with the auto-incremented rule are shown in the Java code snippets in Listing 10.

As shown in Listing 10, the auto-generated rule for `CourseModule.code` is “`M`” `+ v`[6], where `v` is an auto-incremented integral value whose seed is `100 x CourseModule.semester`. That is, for instance, all `Modules` that are taught in semester 1 will have their `Module.code` set to 100, 101, etc. The maximum value of `v` for each semester, that has been used so far is recorded in the `Map`-typed base attribute `CourseModule.currNums`. The key type of this map is a class named `Tuple`, which is used to record the base value(s) used in the computation of `v`. In this case, the base value is the value of the attribute `CourseModule.semester`.

6 in this context, ‘`+`’ is the string concatenation operator

Listing 9 The Java code snippets of `CourseModule` showing how its auto attributes (`CourseModule.id,code`) are maintained

```
@DClass(schema="courseman")
public abstract class CourseModule {
  @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="code",auto=true,type=Type.String,length=6,
      mutable=false,optional=false,derivedFrom={"semester"})
  private String code;

  /**
   * @requires
   *  minVal != null /\ maxVal != null
   * @effects
   *  update the auto-generated value of attribute <tt>attrib</tt>,
   *  specified for <tt>derivingValue</tt>, using <tt>minVal, maxVal</tt>
   */
  @DOpt(type=DOpt.Type.AutoAttributeValueSynchroniser)
  public static void updateAutoGeneratedValue(
      DAttr attrib,
      Tuple derivingValue,
      Object minVal,
      Object maxVal) throws ConstraintViolationException {

    if (minVal != null && maxVal != null) {
      if (attrib.name().equals("id")) {
        int maxIdVal = (Integer) maxVal;
        if (maxIdVal > idCounter)
          idCounter = maxIdVal;

      } else if (attrib.name().equals("code")) {
        String maxCode = (String) maxVal;
        try {
          int maxCodeNum = Integer.parseInt(maxCode.substring(1));

          // current max num for the semester
          Integer currNum = currNums.get(derivingValue);

          if (currNum == null || maxCodeNum > currNum) {
            currNums.put(derivingValue, maxCodeNum);
          }

        } catch (RuntimeException e) {
          throw new ConstraintViolationException(
              ConstraintViolationException.Code.INVALID_VALUE, e,
                   new Object[] {maxCode});
        }
      }
    }
  }
}
```

Listing 10: The Java code snippets of `CourseModule` showing how the value of auto attribute `CourseModule.code` is generated

```
import java.util.Map;
import domainapp.util.Tuple;
//...<omitted>...

  private static Map<Tuple,Integer> currNums = new LinkedHashMap<>();

  // automatically generate a next module code
  private String nextCode(String currCode, int semester)
                                            throws ConstraintViolationException {
    Tuple derivingVal = Tuple.newInstance(semester);
    if (currCode == null) { // generate one
      Integer currNum = currNums.get(derivingVal);
      if (currNum == null) {
        currNum = semester * 100;
      } else {
        currNum++;
      }
      currNums.put(derivingVal, currNum);
      return "M" + currNum;
    } else { // update
      int num;
      try {
        num = Integer.parseInt(currCode.substring(1));
      } catch (RuntimeException e) {
        throw new ConstraintViolationException(
            ConstraintViolationException.Code.INVALID_VALUE, e,
                new Object[] { currCode});
      }

      Integer currMaxVal = currNums.get(derivingVal);
      if (currMaxVal == null || num > currMaxVal) {
        currNums.put(derivingVal, num);
      }

      return currCode;
    }
  }
```

3.6.3. Derived attribute

A derived attribute is a domain attribute whose values are derived (or computed) from those of other domain attributes[7] (these attributes will be referred to as **source attributes**). In DomainAppTool, the developer can specify a domain attribute as derived by setting `DAttr.auto = true` and setting `DAttr.derivedFrom` to an array containing the names of the source attributes. Note that a derived attribute is an auto attribute; the reverse, however, is not true.

For example, Listing 11 below shows how the attribute `CourseModule.code` is specified as derived from the attribute `CourseModule.semester`.

7 FIT students should refer to the relevant PPL lecture(s) on this

Listing 11: The Java specification of the derived attribute `CourseModule.code` from the source attribute `CourseModule.semester`

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

  @DAttr(name="code",id=true,auto=true,type=Type.String,length=6,
      mutable=false,optional=false,derivedFrom={"semester"})
  private String code;

  @DAttr(name="semester",type=Type.Integer,optional=false,
      length= 2,min=1)
  private int semester;
}
```

Multi-source derived attribute

A multi-source derived attribute is the attribute whose values are derived from more than one source attributes. An example of this is shown in Listing 12, which shows how attribute `Enrolment.finalMark` is derived from two source attributes: `Enrolment.internalMark` and `Enrolment.examMark`. Attribute `Enrolment.finalMark` is used to compute yet another derived attribute `Enrolment.finalGrade`.

Listing 12: The Java specification of the multi-source derived attribute `Enrolment.finalMark` (derived from both `internalMark` and `examMark`)

```
@DClass(schema="courseman")
public class Enrolment implements Comparable {
  private static final String AttributeName_InternalMark = "internalMark";
  private static final String AttributeName_ExamMark = "examMark";
  private static final String AttributeName_FinalMark = "finalMark";

  @DAttr(name=AttributeName_InternalMark,type=Type.Double,length=4,
    optional=true,min=0.0) private Double internalMark;

  @DAttr(name=AttributeName_ExamMark,type=Type.Double,length=4,
    optional=true,min=0.0) private Double examMark;
  @DAttr(name="finalGrade",auto=true,type=Type.Char,
    mutable=false,optional=true,length=1
  /* Note: no need to do this: derivedFrom={"internalMark,examMark"}
   *  because finalGrade and finalMark are updated by the same method and this is
   *  already specified by finalMark (below)
   */
  )
  private char finalGrade;

  @DAttr(name=AttributeName_FinalMark,auto=true,type=Type.Integer,
      mutable=false,optional=true,serialisable=false,
      derivedFrom={AttributeName_InternalMark, AttributeName_ExamMark})
  private Integer finalMark;
}
```

Note from Listing 12 that, in principle, attribute `finalGrade` is also a multi-source derived attribute, derived from the same two source attributes. However, the specification of this attribute needs not specify a value for property `DAttr.derivedFrom`. The reason for this, as will become clear in the next subsection, is because the values of this attribute are computed and maintained by the same operation that maintains attribute `finalMark`.

3.6.4. Maintaining a derived attribute

Similar to auto attributes, the domain class that contains a derived attribute must be defined with an operation that implements suitable domain-specific rules for generating and maintaining values for the attribute. To ease discussion, let us refer to this operation as **value-deriving operation**. Depending on the number of source attributes involved, extra steps need to be taken to ensure that values of the source attribute(s) are maintained correctly. More specifically, if the derived attribute is derived from a single source attribute (e.g. `CourseModule.code`) then there are no additional design requirements for the deriving operation. If the derived attribute (e.g. `Enrolment.finalMark`) is derived from multiple source attributes then the following additional design rules must be observed:

1. **header rule**:

   the header of the value deriving operation must be specified with two helper annotations `DOpt` *and* `AttrRef` *as follow:*

`@DOpt(type=DerivedAttributeUpdater)`

`@AttrRef(name="`*derived_attribute_name*`")`

(where *derived_attribute_name* is the name of the derived attribute involved)

2. **state rule**:

   the previous value of the derived attribute must be recorded (at least temporarily) in a state cache (ref. class: `StateHistory`*)*

3. **operation rule**:

   the setter operations of the source attributes must not directly invoke the value deriving operation (this operation is invoked automatically by the tool)

Listing 10 given earlier shows how `CourseModule.code` is maintained by the value deriving operation named `nextCode`. This is a recommended naming convention for this type of operation. Listing 13 below, on the other hand, shows the Java specification and code snippets of the domain class `Enrolment`, which show how the attribute `finalMark` and the two source attributes are maintained.

Lines 6 and 17 of Listing 13 show the definition of the state cache (var. `stateHist`). It is a `Map` from `String` to `Object`, which only temporarily caches the values of `Enrolment`'s derived attributes from the time that they are recorded to the time of their first read. As shown further down in the listing, the value of `Enrolment.finalMark` is recorded in this cache as part of operation `updateFinalMark` and this value is retrieved (and removed) from the cache during the invocation of operation `getFinalMark(boolean)`.

The codes of the two operations `updateFinalMark` and `getFinalMark(boolean)` are self-explanatory, except for the following points. First, method `updateFinalMark` is declared `public`, so that the tool can invoke it directly. Second, operation `getFinalMark` throws the unchecked exception named `IllegalStateException`, which occurs when the argument `cached = true` but no value of `finalMark` is found in the state cache.

Listing 14 shows how operation `getFinalMark(boolean)` is used as part of the link-updater method of class `Student`. This is where the cached value of `Enrolment.finalMark` is first read and this value is used to compute the change in `finalMark`. This change is then used to compute the value for derived attribute `Student.averageMark`.

Listing 13: Java specification and code snippets of `Enrolment`, showing how `internalMark`, `examMark`, `finalMark` and `finalGrade` are maintained

```java
import domainapp.util.cache.StateHistory;

@DClass(schema="courseman")
public class Enrolment implements Comparable {

  private StateHistory<String, Object> stateHist;

  @DOpt(type=DOpt.Type.DataSourceConstructor)
  public Enrolment(@AttrRef("id") Integer id, @AttrRef("student") Student s,
 @AttrRef("module") CourseModule m, @AttrRef("internalMark") Double internalMark,
 @AttrRef("examMark") Double examMark, @AttrRef("finalGrade") Character finalGrade)
 throws ConstraintViolationException {
      this.id = nextID(id);
      this.student = s;
      this.module = m;
      this.internalMark = (internalMark != null) ? internalMark.doubleValue()
          : null;
      this.examMark = (examMark != null) ? examMark.doubleValue() : null;

      stateHist = new StateHistory<>();
      updateFinalMark();
    }

    public void setInternalMark(Double mark) {
      // update final grade = false: to keep the integrity of its cached value
      setInternalMark(mark, false);
    }

    public void setInternalMark(Double mark, boolean updateFinalGrade) {
      this.internalMark = mark;
      if (updateFinalGrade)
        updateFinalMark();
    }

    public void setExamMark(Double mark) {
      // update final grade = false: to keep the integrity of its cached value
      setExamMark(mark, false);
    }

    public void setExamMark(Double mark, boolean updateFinalGrade) {
```

```
    this.examMark = mark;
    if (updateFinalGrade)
      updateFinalMark();
  }

  @DOpt(type=DOpt.Type.DerivedAttributeUpdater)
  @AttrRef(name=AttributeName_FinalMark)
  public void updateFinalMark() {
    // updates both final mark and final grade
    if (internalMark != null && examMark != null) {
      double finalMarkD = 0.4 * internalMark + 0.6 * examMark;
      // cache final mark
      stateHist.put(AttributeName_FinalMark, finalMark);
      // round the mark to the closest integer value
      finalMark = (int) Math.round(finalMarkD);

      if (finalMark < 5)
        finalGrade = 'F';
      else if (finalMark == 5)
        finalGrade = 'P';
      else if (finalMark <= 7)
        finalGrade = 'G';
      else
        finalGrade = 'E';
    }
  }

  public int getFinalMark() {
    return getFinalMark(false);
  }

  public int getFinalMark(boolean cached) throws IllegalStateException {
    if (cached) {
      Object val = stateHist.get(AttributeName_FinalMark);
      if (val == null)
        throw new IllegalStateException("Enrolment.getFinalMark: cached value is null");
      return (Integer) val;
    } else {
      if (finalMark != null)
        return finalMark;
      else
        return 0;
    }
  }
}
```

Listing 14: A partial Java code of `Student` showing how operation `Enrolment.getFinalMark(boolean)` is used

```
@DClass(schema="courseman")
public class Student {
  private double averageMark;

  @DOpt(type=DOpt.Type.LinkUpdater)
  public boolean updateEnrolment(Enrolment e) throws IllegalStateException {
    // recompute 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;
  }
}
```

3.7. Advanced design issues

In this section, we discuss the relatively advanced issues concerning the design of some special domain class members. We first discuss in Section 3.7.1 the concept of candiate identifier. We then discuss in 3.7.2 how to design enum data type. Next, in Section 3.7.3 we discuss how to design domain-specific exceptions in the software. Afte that, we discuss in Section 3.7.4 the design of object count. Then in Section 3.7.5, we explain the design of constructor operation in a domain class hierarchy.

3.7.1. Candidate identifier

Most of the domain class design examples so far use a primary auto-generated identifier attribute. While this design is efficient, it has one limitation in that the attribute values are not easy for the user to comprehend. This is the case when we need to present the identifier values on a combo or list box for the user to choose. To overcome this, we need to be able to specify some other domain attributes of the domain class to become the candidate identifier. These attributes have the same property as the primary identifier that they uniquely identify the domain objects.

To support this design, DOMAINAPPTOOL provides two additional `DAttr` properties:

1. `cid: boolean`

   specifies a single attribute to become the candidate identifier

2. `ccid: String`

   specifies a group of attributes to become the candidate composite identifier. The values of this property in all attributes of the group must be the same.

   Note: This feature is not yet supported!

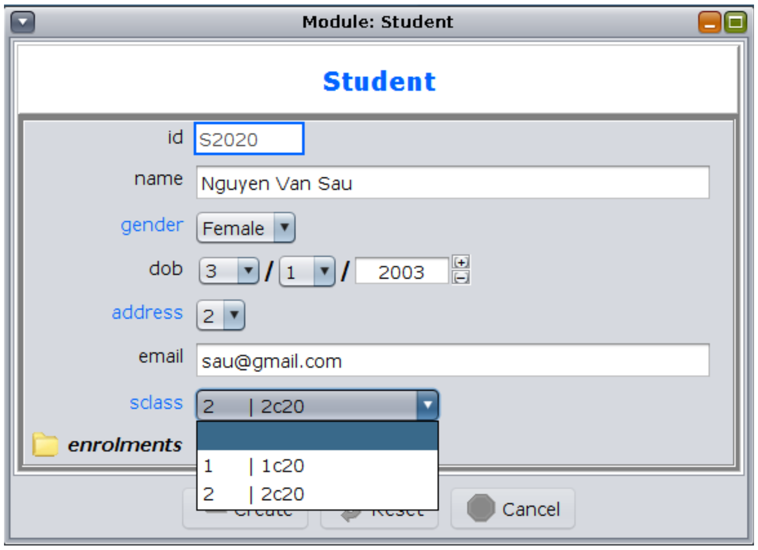
Listing 15 shows how property `DAttr.cid` is used to set the attribute `SClass.name` to be the candidate identifier.

Listing 15: Specifying a candidate id attribute (`cid`) for `SClass`

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

Bounding candidate identifier to a view field

Candidate identifier is useful for displaying information about the domain objects of the owner class on a bounded view field. This binding us performed automatically by DOMAINAPPTOOL. Figure 12 shows an example of how the candidate identifier `SClass.name` (defined above) is bounded to the view field `Student.sclass` on the `Student`



id: S2020
name: Nguyen Van Sau
gender: Female
dob: 3 / 1 / 2003
address: 2
email: sau@gmail.com
sclass: 2 | 2c20
1 | 1c20
2 | 2c20

Figure 12: Bounding the candidate identifier `SClass.name` to a view field.

form. Instead of displaying only the numeric id numbers, this field now displays also the `SClass.name` values. These values are easier for user to understand than the `id` values.

3.7.2. Using enum as data type

In many cases where a domain field's range is taken from a small set of pre-defined values, it is best to represent these values by a enum and use this enum as the field's data type. Among the common examples are gender (male, female), daily sessions (morning, afternoon, evening) and season (spring, summer, fall, winter).

Because enum is also a class, we can use it as a field's data type the same way as we would with a normal domain class. However, turning an enum into a domain class is much simplied as described below:

- add a method called `getName`
- specify this method with an id-typed `DAttr` for the enum value as follows: `DAttr.name="name"`, `DAttr.type=DAttr.Type.String`, `DAttr.id=true`
- in the method body, call the `name()` method of the enum to return the value

Listing 16 illustrates the above design with an enum called `Gender`. This enum represents three common gender values of a person. Note how the value of property `DAttr.length` is set to 10, which is appropriate for the three gender values.

Listing 16: Designing the `Gender` enum as a domain class.

```
/**
 * @overview Represents the gender of a person.
 * @author Duc Minh Le (ducmle)
 */
public enum Gender {
  Male,
  Female,
  Others;
  @DAttr(name="name", type=DAttr.Type.String, id=true, length=10)
   public String getName() {
     return name();
    }
}
```

3.7.3. Handling domain-specific software errors

As a software framework, DomainAppTool provides not only pre-defined exceptions for the most common error cases but also an extension mechanism for domain software to define their own domain-specific exceptions. For efficiency, DomainAppTool defines two generic exception classes that take an error code as input. Thus, domain software can define their own exception codes and use them in throwing their exceptions.

Follow these steps to define and use domain-specific exception codes in a software:

1. Define an enum that implements the interface `domainapp.basics.util.InfoCode`. Use this enum to define the exception codes
2. Define each enum value with this format:

   `ENUM_VALUE("...{0}...{1}...")`

   where `{0}, {1}, ...` are template variables that will be replaced by run-time values
3. In the code that needs to throw a domain-specific exception, use the `ConstraintViolationException` with a suitable exception code that you defined above.

Listing 17 shows the implementation of an example domain exception code enum, named `DExCode`. This class is available in the courseman project and use can use it as the template to define your own enum. Notice that the code block that begins at the declaration of the private field `text` needs not be changed. You only need to change the constructor method name if you decide to change the enum name. The main task to do is to define the enum values in the block above this code block. For simplicity, the listing shows only one enum value named `INVALID_DOB`, which captures the error message for the case that `Student.dob` is given an invalid date. In this case, the template variable `{0}` of `INVALID_DOB` will be received by the run-time value of `Student.dob`, so that it is also displayed to the user as part of the exception message.

Listing 17: An example domain exception code implementation

```
import java.text.MessageFormat;
import domainapp.basics.util.InfoCode;
/**
 * @overview Capture domain-specific exception codes.
 *   ... omitted...
```

```java
 * @author Duc Minh Le (ducmle)
 */
public enum DExCode implements InfoCode {

  /**
   * 0: date of birth
   */
  INVALID_DOB("Date of birth {0} is not a valid date"),
  // other enum values go here
  ;

  /**
   * THE FOLLOWING CODE (EXCEPT FOR THE CONSTRUCTOR NAME) MUST BE KEPT AS IS
   */
  private String text;

  /**The {@link MessageFormat} object for formatting {@link #text}
   * using context-specific data arguments
   */
  private MessageFormat messageFormat;

  private DExCode(String text) {
    this.text = text;
  }

  @Override
  public String getText() {
    return text;
  }

  @Override
  public MessageFormat getMessageFormat() {
    if (messageFormat == null) {
      messageFormat = new MessageFormat(text);
    }
    return messageFormat;
  }
}
```

Listing 18 shows an example of how to use a domain-specific exception code (`DExCode`) in a domain class. In this case, DExCode.INVALID_DB is used in the method setDob to throw a ContraintViolationException when the input dob is before some pre-defined minimum date. Notice how this dob is passed into the exception object as an argument. As mentioned above, the run-time value of this variable will replace the template variable {0} in the exception message defined by DExCode.INVALID_DB.

Listing 18: Using DExCode.INVALID_DB in the Student class

```java
public void setDob(Date dob) throws ConstraintViolationException {
    // additional validation on dob
    if (dob.before(DToolkit.MIN_DOB)) {
      throw new ConstraintViolationException(DExCode.INVALID_DOB, dob);
    }

    this.dob = dob;
  }
```

3.7.4. Maintaining the object count

The computation of Student.averageMark shown earlier in Listing 14 uses a variable named enrolmentCount. This is also a derived attribute of Student, which plays a specific role in recording the current number of Enrolment domain objects that are associated to a given Student object. This object count is primarily used to validate the cardinality constraints of the one-many association between Student and Enrolment.

The question is: Why do we need to define this attribute if it can simply be derived from Student.enrolments? The answer is that for performance reason, the tool does not load all the associated Enrolment objects and put them in Student.enrolments. Therefore, we cannot rely on this attribute for computing object count. What the tool does instead is to load the object count from the data source and set it into the object count variable for subsequent use.

Listing 19 gives details on how the attribute Student.enrolmentCount is defined and maintained. The code is rather self-explanatory except for the two getter and setter operations of enrolmentCount. The basic idea is to use the setter operation to update enrolmentCount such that it is equal to the number of the Enrolment objects *actually* associated to a given Student. Then, whenever an Enrolment object is added to or removed from the current Student (via the link-adder and link-remover operations, respectively), we update enrolmentCount.

The two operations setEnrolmentsCount and getEnrolmentsCount are used as setter and getter for enrolmentCount. They are specified with the helper annotation @DOpt(type=LinkCountGetter). Note that the setter operation is an optimisation feature needed to facilitate the recording of the object count. This feature is introduced at the cost of partially breaking the domain constraint rule of the object count attribute! Why is this setter slightly controversial? Because, technically speaking, attribute enrolmentCount is an auto attribute and should thus be immutable. However, the setter is only invoked by the tool (when it reads the object count value from the data source) and never by the software directly.

Listing 19: Java code snippets of Student showing how attribute enrolmentCount is defined and maintained

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

  private int enrolmentCount;



  @DOpt(type=DOpt.Type.DataSourceConstructor)
  public Student(@AttrRef("id") String id, @AttrRef("name") String name,
      @AttrRef("dob") String dob, @AttrRef("address") City address,
      @AttrRef("email") String email, @AttrRef("sclass") SClass sclass)
       throws ConstraintViolationException {
    // generate an id
    this.id = nextID(id);
    // assign other values
    this.name = name;
    this.dob = dob;
    this.address = address;
    this.email = email;
    this.sclass = sclass;
    this.enrolments = new ArrayList();
    enrolmentCount = 0;
    averageMark = 0D;
  }
```

```
@DOpt(type=DOpt.Type.LinkAdderNew)
public boolean addNewEnrolment(Enrolment e) {
  enrolments.add(e);
  enrolmentCount++;
  computeAverageMark();
  // no other attributes changed (average mark is not serialisable!!!)
  return false;
}

@DOpt(type=DOpt.Type.LinkAdderNew)
public boolean addNewEnrolment(Collection<Enrolment> enrols) {
  enrolments.addAll(enrols);
  enrolmentCount += enrols.size();
  computeAverageMark();
  // no other attributes changed (average mark is not serialisable!!!)
  return false;
}

@DOpt(type=DOpt.Type.LinkRemover)
public boolean removeEnrolment(Enrolment e) {
  boolean removed = enrolments.remove(e);
  if (removed) {
    enrolmentCount--;
    computeAverageMark();
  }
  // no other attributes changed
  return false;
}

public void setEnrolments(Collection<Enrolment> en) {
  this.enrolments = en;
  enrolmentCount = en.size();
  computeAverageMark();
}
```

```
  @DOpt(type=DOpt.Type.LinkCountGetter)
  public Integer getEnrolmentsCount() {
    return enrolmentCount;
  }

  @DOpt(type=DOpt.Type.LinkCountSetter)
  public void setEnrolmentsCount(int count) {
    enrolmentCount = count;
  }

  /**
   * @effects
   *  computes {@link #averageMark} of all the {@link Enrolment#getFinalMark()}s
   *  (in {@link #enrolments}.
   */
  private void computeAverageMark() {
    if (enrolmentCount > 0) {
      double totalMark = 0d;
      for (Enrolment e : enrolments) {
        totalMark += e.getFinalMark();
      }
      averageMark = totalMark / enrolmentCount;
    } else {
      averageMark = 0;
    }
  }
}
```

3.7.5. Modelling a domain class hierarchy

A domain class hierarchy is modelled using the standard type hierarchy design method[8], plus the following extra rules that are required by DomainAppTool.

Parameter list rule

CR7. *In the data source constructor, the parameters that are used to set the values of mandatory, subtype-specific domain attributes must be placed after those of the super-*

8 Students should refer to the prerequisite OOPL knowledge.

type in the parameter list of the constructor header.

Base constructor rules

CR8. *A subtype must define the necessary base constructors to invoke those of its supertype.*

An example type hierarchy used in the CourseMan software is shown in Figure 1 and Appendix 1. This hierarchy consists of an `abstract` super-type named `CourseModule` and two sub-types named `CompulsoryModule` and `ElectiveModule`. Among these, only `ElectiveModule` has a subtype specific attribute named `deptName`.

Domain constraint extension

When we need to specify in a sub-type a domain constraint extension for an inherited domain attribute, we need to:

- override the observer method of the attribute. This method simply needs to invoke the supertype's method.
- specifies the extension using a `DAttr` that is attached to this method. This `DAttr` must preserve the property values of the `DAttr` in the supertype.

For example, we may want to specify that attribute `ElectiveModule.semester` must not be lower than 3. This is to enforce a rule that elective modules are not taught in the first year. Listing 20 shows how this is achieved. Specifically, the `DAttr` attached to the observer method `getSemester` inherits all the property values from the `DAttr` of the attribute `CourseModule.semester`. In addition, property `DAttr.min` is set to 3 instead of 1.

Listing 20: Domain constraint extension for attribute `ElectiveModule.semester`

```
@DClass(schema="courseman")
public class ElectiveModule extends CourseModule {
  @DAttr(name="semester",type=Type.Integer,length=2,optional=false,min=3)
  @Override public int getSemester() { return super.getSemester(); }
}
```

Exercises

Ex1. Change the domain attribute `City.name` such that it is optional. Try creating a new `City` object and update an existing one without specifying a value for attribute `name`. Observe what happens.

Ex2. Change the domain attribute `Student.address` such that it is not optional. Create two new `Students` with and without specifying an address and observe what happen.

Notice, in particular, how the value of the data field `City.student` is updated automatically.

Ex3. Change the domain attribute `Student.email` to immutable. Observe what happens when you (1) create a new `Student` object and (2) edit an existing `Student` object.

After you've completed this exercise, change `Student.email` back to its original design.

Ex4. Change the domain attribute `Enrolment.examMark` to immutable. Observe what happens when you create a new or edit an existing `Enrolment` object using the stand-alone `Enrolment` object form as well as using the `Enrolment`'s sub-form of the `Student`'s object form.

After you've completed this exercise, change `Enrolment.examMark` back to its original design.

Ex5. Change the max cardinality of `Student` in the association with `SClass`, named `class-has-student`, from 25 to 3. Try creating more than three students in one class and observe what happens.

Ex6. Change the domain attributes `Enrolment.internalMark` and `Enrolment.examMark` such that they both have the max value of 10.0.

Try creating a few `Enrolments` with values of the above attributes both inside and outside the allowed value range. Observe what happens.

Ex7. Change the domain attributes `CourseModule.semester` and `CourseModule.credits` such that their value ranges are [1,8] and [1,10], respectively.

Try creating a few `CourseModules` with values of the above attributes both inside and outside the respective value ranges. Observe what happens.

Ex8. Change the domain attribute `SClass.name` to reduce its length from 20 to 5.

Create a few `SClass` objects with values of the above attribute both within and exceeding the length. Observe what happens.

Ex9. Update the class `CompulsoryModule` to add a new domain attribute named `lecturerName`, whose data type is `String`. This attribute is to model the name of the lecturer who is assigned to teach a compulsory module.

Note:

(a) You need to delete the database table of `CompulsoryModule` with the command `deletedomainmodel` and re-run the software.

(b) You need to reconfigure the software in order to include the new attribute on the GUI.

Ex10. Update attribute `ElectiveModule.credits` to specify a rule that it must take values in the range [3,5]. Try creating a few `ElectiveModules` with values of attribute `credit` both inside and outside the value range. Observe what happens.

Ex11. Change the cardinality of `Student` in the association with `City` from 1:1 to 1:* and that `Student` is no longer the determinant of the association.

Note: Use the cardinality constant `MetaConstants.CARD_MORE` for the "*" in "1:*".

Ex12. Implement the associative attribute `CourseModule.enrolments`. Ensure to add suitable operations.

Ex13. Define domain-specific exceptions for all of the violations of the domain constraints reported in the previous exercises.

Ex14. Change the domain attribute `Student.averageMark` such that it becomes a serialisable domain attribute.

Note: You will need to reconfigure the software in order to update the database schema.

Ex15(*). Update the class `SClass` to add a new derived attribute named `averageMark` which is computed from `averageMark` of all `Students` in an `SClass`.

Note: to run the software with the design changes, first use the command `deleteconfig` to delete the existing configuration, then re-run with the `configure` command.

Ex16(*). Extend Ex15 to turn `SClass.averageMark` into a serialisable domain attribute.

Ex17. (*For FIT students who have done the SE1 course module*) Convert the FSIS software that you did in the SEG module into a software that is run in DomainAppTool.

Chapter 4. Managing the domain model instances

Once the domain model has been created, it can be used in the software to create domain objects. These domain objects form what is called a **domain model instance**. There are two methods for creating a software from the domain model. The first and most basic method is to create a software that directly manipulates the domain model to create its instances. We call this software **model-driven software**. This method does not require the software GUI and is useful for independently testing the domain model. The second and primary method is to create a **GUI-based software** that consists of a GUI for manipulating the objects. We will discuss this method in the next chapter.

This chapter focuses on the first method. It presents a domain model API that is used to programmatically manipulate the model and its instances.

4.1. Model-driven Software

We explain in this section how to create a basic software, called **model-driven software**, that directly works with the domain model API. The software class is named `DomSoftware`. All operations on the domain model are performed through a `DomSoftware` object. The following code creates a default `DomSoftware`, which uses a JavaDB's embedded relational database.

```
DomSoftware sw = SoftwareFactory.createDefaultDomSoftware();
```

Each time the software is run, it needs to be initialised first by invoking the method `init` as follows:

```
sw.init();
```

In the next section, we will assume the existence of the software variable `sw`.

4.2. Domain Model API

The purpose of this API is to provide a programming interface for software to manipulate the domain model elements and its instances. The API is organised around a well-known set of four basic object manipulation operations: create, read, update and delete (CRUD)[9]. We first explain the CRUD operations on the domain model. After that, we describe the CRUD operations on the instances. This in turn is broken down into CRUD operations on the domain objects and the object links.

9 URL: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

4.2.1. Creating a domain model

This basically involves creating each domain class in the model and registering this class to the system. The guidelines for how to create a domain class was explained in the previous chapter.

Registering a class to the system involves initialising an object pool for holding objects and initialising an object store for it in the underlying data source. Registering a class also automatically registering the classes that are associated directly and indirectly to it. This is to ensure that the domain objects and their links are created correctly. We call the domain classes that are associated in this way a **model fragment**. To materialise a model fragment requires only to specify the core domain class of the fragment. If the underlying data source is a relational database, then an object store of a domain class is a table in this database.

For instance, the following code both registers the model fragment concerning the `Student` class to the system:

```
Class[] domFrag = {
   Student.class
};
sw.addClasses(domFrag);
```

This fragment includes of the following domain classes that are associated directly and indirectly to `Student`:

- direct association: `Gender`, `City`, `SClass`, `Enrolment`
- indirect association: `CourseModule` (via `Enrolment`)

4.2.2. Reading a domain model

Once registered into the system, the domain model is managed by the model manager component. The domain model and its instances can be displayed to the standard output using the following code:

```
boolean displayFqn = false;
sw.printDomainModel(displayFqn);
```

4.2.3. Updating a domain model

Because the domain model keeps references to the domain class objects, any changes to the domain class are immediately reflected in the domain model. If a domain class is no longer used and to be removed from the system, use the following code to properly unregister it. This code also delete the model fragment concerning this class (together with the underlying object stores):

```
Class[] toDelete = {Student.class};
try {
  sw.deleteDomainModel(toDelete);
} catch (DataSourceException e) {
  e.printStackTrace();
}
```

4.2.4. Deleting a domain model

There are two methods of delete a domain model: (1) deleting one fragment at a time, (2) delete the entire domain model. Note that deleting a domain model also deleting all domain classes and their object stores in the data source.

Similar to model fragment registration, to delete a model fragment requires only to specify the core domain class of that fragment. The code example in Section 4.2.3 demonstrates this.

To delete the entire domain model requires to specify the domain model name. This name is taken from the value of `@DClass.schema` property of the domain classes in the model. The following code demonstrates.

```
String modelName = sw.getDomainModelName(Student.class);
try {
  sw.deleteDomainModel(modelName);
} catch (DataSourceException e) {
  e.printStackTrace();
}
```

4.2.5. Creating a domain object

This involves instantiating the domain object (by invoking a suitable constructor) and adding this object to the system. The latter action consists in adding the object to the object pool and the object store of the object class. The following code demonstrates how to create a `City` object.

```
City obj = new City(1, "Hanoi");
sw.addObject(City.class, obj);
```

4.2.6. Reading domain objects

There are two main methods for reading objects: (1) using an attribute-value pair and (2) using a query (for more complex search requirement).

Reading objects using a name-value pair

The following code demonstrates to read (a.k.a retrieve) a `Student` object whose `id` equals "S2020". The result is a parameterised `Collection` and can be displayed to the standard output using the helper method `sw.printObjects`.

```
Collection<Student> objects = sw.retrieveObjects(Student.class,
   Student.A_id, Op.EQ, "S2020");
sw.printObjects(cls, objects);
```

A special case is when the attribute is the identifier attribute named "`id`". In this case, using code can invoke the short-cut method `sw.retrieveObjectById`, as demonstrated in the following code:

```
Student s = sw.retrieveObjectById(Student.class, "S2020");
```

Reading objects using a query

Reading objects using a query requires using the query manager class named `QRM`. We will this class and the query API in more detail later in this book. In this section, we demonstrate using a simple example, which queries the data source for objects whose names match a given name pattern.

```
QRM qrm = QRM.getInstance();
// create query
String namePattern = "%"+name+"%";
Query q = QueryToolKit.createSearchQuery(
    qrm.getDsm(),
    Student.class,
    new String[] {Student.A_name},
    new Op[] {Op.MATCH},
    new Object[] {namePattern});

System.out.printf("Querying students with name matching '%s'%n", namePattern);
Map<Oid, Student> result = qrm.getDom().retrieveObjects(Student.class, q);
```

4.2.7. Updating a domain object

Given that an object exists in the object store, updating it requires saving the changes back to the data store. This is achieved by invoking the method `sw.updateObject`. The following code demonstrates.

```
Student s = sw.retrieveObjectById(Student.class, id);
if (s != null) {
  System.out.printf("Updating object%n%s%n", s);
  sw.updateObject(Student.class, s,
      new String[] {
          Student.A_email, Student.A_address},
      new Object[] {
          "leminhduc@gmail.com",
          sw.retrieveObjectById(City.class, 2)
      });
  System.out.printf("... after:%n%s%n", s);
}
```

4.2.8. Deleting a domain object

Given that an object exists in the object store, deleting it requires removing its record in the object store. This is achieved by invoking the method `sw.deleteObject`. The following code demonstrates.

```
Student s = sw.retrieveObjectById(Student.class, id);
if (s != null) {
  System.out.printf("Deleting object%n%s%n", s);
  sw.deleteObject(s, Student.class);
}
```

4.3. UI Software

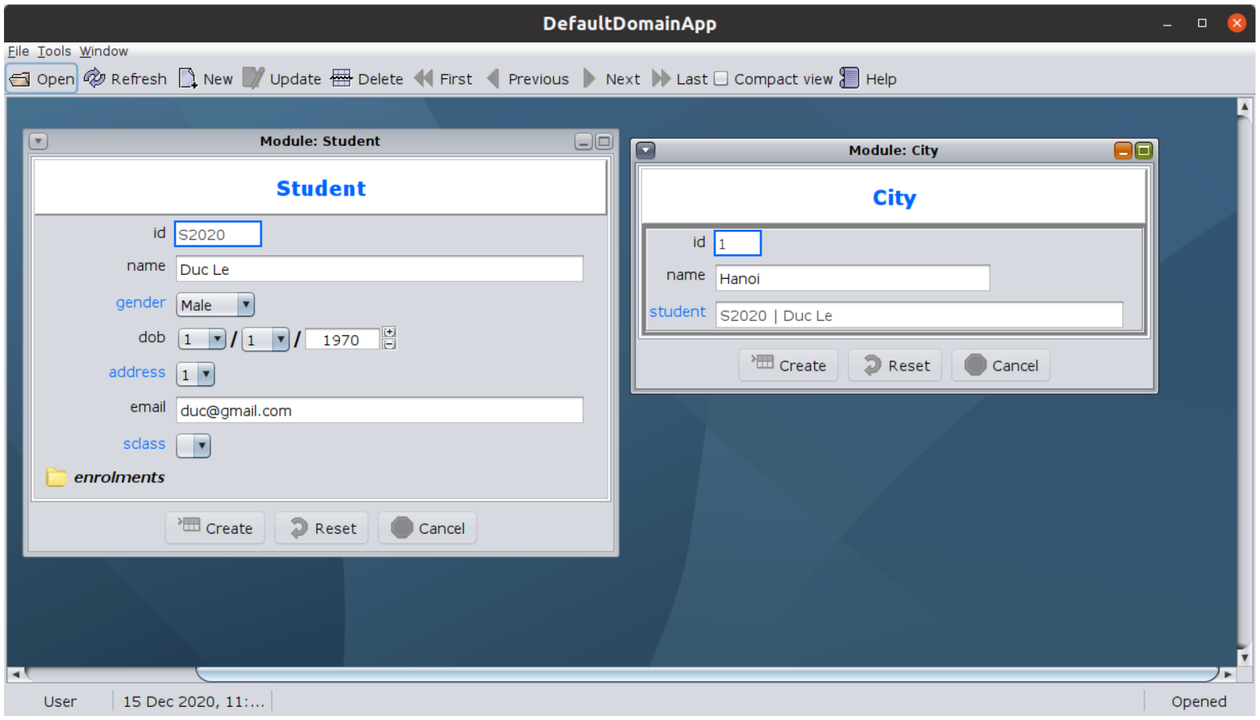
UI software is a subtype of class `DomSoftware` that provides a GUI for user to perform tasks more conveniently on the domain model. This GUI helps simplify the interface to the domain model API. The next chapter will discuss the design architecture of the UI software in more detail.

To create a UI software requires two steps:

1. Initialises a domain model
2. Use `SoftwareFactory` to create a software object
3. Invoke the method `sw.run` to run the software with the model

The following code listing demonstrates this with the `CourseMan` example. The three steps are labelled clearly in the listing. Figure 13 shows the software's GUI that is presented to the user. The entire software GUI is automatically generated by the tool.

```
/**
 * @effects
 *   create and run a UI-based {@link DomSoftware} for a pre-defined model.
 */
public static void main(String[] args){
  // 1. initialise the model
  Class[] model = {
      CourseModule.class, CompulsoryModule.class, ElectiveModule.class,
      Enrolment.class, Student.class, City.class,
      SClass.class,
      // reports
      StudentsByNameReport.class,
      StudentsByCityJoinReport.class
  };
  // 2. create UI software
  DomSoftware sw = SoftwareFactory.createUIDomSoftware();
  // 3. run it
  try {
    sw.run(model);
  } catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }
}
```



id	S2020
name	Duc Le
gender	Male
dob	1 / 1 / 1970
address	1
email	duc@gmail.com
sclass	
enrolments	

id	1
name	Hanoi
student	S2020 | Duc Le

15 Dec 2020, 11:...

Figure 13: The automatically-generated UI software for CourseMan.

Exercises

Ex1. What are the different ways of creating a domain model?
Ex2. Write a program to print a domain model fragment to the standard output.
Ex3. Write a program to print to the standard output the `Students` who live in Hue.
Ex4. Write a program to create a new `Student` who is enrolled in a `CompulsoryModule` named “SEG”. Display the object store of `Student`.
Ex5. Write a program to change all `Students` in the data source so that half of them live in Hanoi and the other half live in HCM city.
Ex6. Write a program to search in the data source for `Students` studying in the class named “2c15” who are 20 years of age. Display the result to the standard output.

Chapter 5. Automatically generating a software from the domain model

A key benefit of carefully designing the domain model with meta-attributes, as explained in the previous chapter, is that the resulted model can be processed automatically by the underlying object-oriented programming language platform to generate the software. This software consists of the domain model at its core, a GUI that presents the domain classes for the user to easily interact and work with their objects and a data storage system that is responsible for permanently storing the domain objects.

The DomainAppTool was developed with an aim to help automate this process in the Java programming language. Conceptually, it takes as input a Java-based domain model, processes the domain classes to generate the GUI and a relational database schema, and automatically run the software with these components. In this section, we will explain our proposed method and demonstrate it using the CourseMan domain model.

5.1. Process overview

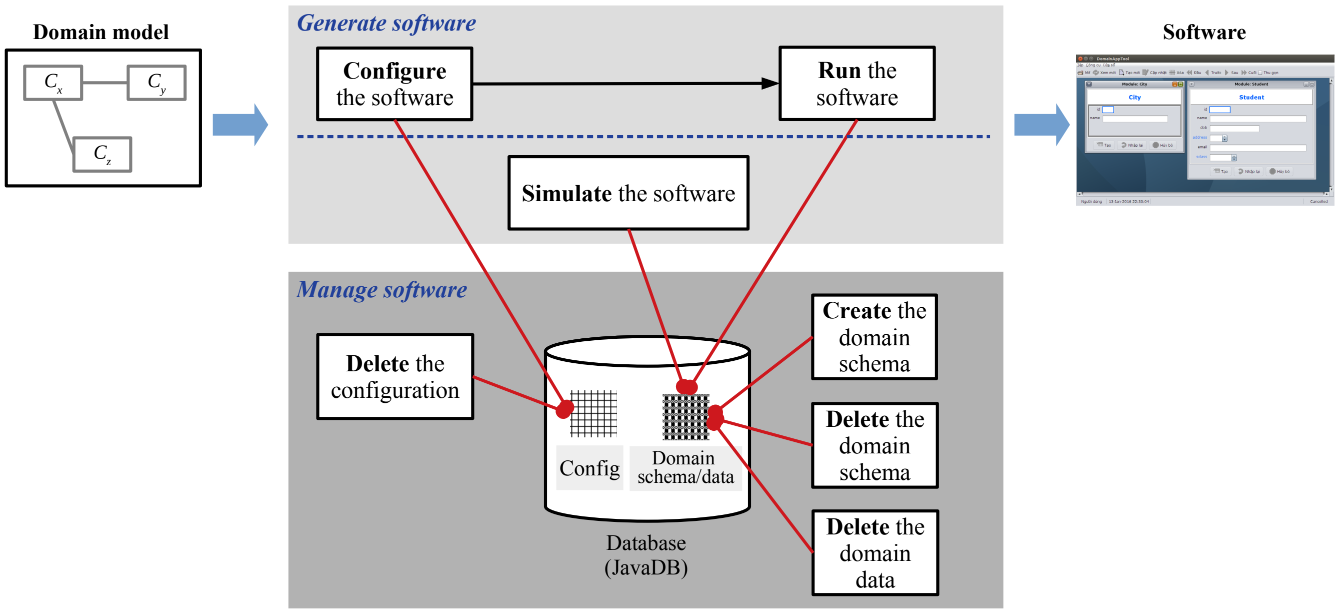


Figure 14: Generating and managing software in DomainAppTool.

The DomainAppTool's process consists of two sub-processes: (1) generate software and (2) manage software. The entire process is summarised in Figure 14. Each sub-process consists of one or more tasks, called **software tasks**. Each task is realised (and thus executed) by a DomainAppTool command. We provide below a brief description of the sub-processes and their tasks. The subsequent subsections will discuss how to perform each task.

Both sub-processes and their tasks operate on a database that consists of two separate schemas: one for storing the software configuration, the other is for storing the domain data (i.e. the domain objects). We call the first schema **configuration schema** (abbrv. **Config** in Figure 14) and the second schema **domain schema**. Both sub-processes operate on these two schemas.

The first sub-process (depicted in the top box of Figure 14) consists of three tasks, two of which are performed in sequence and form the standard software generation scenario. The third task, called "Simulate the software", provides an alternative generation scenario, which combines software configuration and execution together. In this scenario, software configuration is generated in memory is not stored in the Config schema. This task is therefore mainly used during the development phase to quickly test the software design.

The second sub-process (depicted in the bottom box of Figure 14) consists of the following tasks:

1. Delete the configuration
2. Create the domain schema
3. Delete the domain schema
4. Delete the domain data

The first task operates on the Config schema of the database. The other tasks operate on the domain schema and the domain data.

DomainAppTool provides the developer with two interfaces through which she can perform the software tasks. Sections 5.2 and 5.3 will discuss the first interface, which is a command-line interface that is defined through a generic executable class. Section 5.4 explains the second interface, which is a programming interface that is easier to run and customise.

5.2. Using the command line

In this section we explain a basic command line interface for executing the software tasks. This interface requires the developer to enter the command directly on the command line interface of the operating system. In Windows, this is the `Cmd` program; while in Linux, this is the `Terminal` program (or a variant of this).

Given that readers are already familiar with the Java's `java` command, the basic instruction is to run this command for the DomainAppTool's binary (file: `domainapptool.jar`), specifying as input a list of the (fully qualified) names of the domain classes that make up the software's domain model. We will call this domain model the **domain model binary**. This binary is either a directory that contains the byte codes of the domain classes or a jar file (e.g. `courseman.jar`). The class path setting of the command must contain this domain model binary and the necessary libraries that are provided as part of the tool's distribution.

5.2.1. Preparation

We summarise below the steps to prepare the DomainAppTool and the domain model binary for execution.

1. Create a *program directory* for the software, e.g. `/home/user/CourseMan` (for Linux) or `C:\CourseMan` (for Windows)
2. Decompress the DomainAppTool distribution file to the program directory
3. Copy the domain model binary to the program directory

All the commands that will be explained in the subsequent sections are executed using the program directory as the current working directory. Further, we will refer to `CourseMan` as the *default software*.

5.2.2. Configuring the software

This task is performed when the software is generated for the first time or when, for some reasons, its configuration needs to be reinitialised. The latter typically occurs when the domain model has been changed in such a way that affects the object forms of some domain classes in the model.

To configure the software, we use the command named `configure`. To illustrate this command, let us consider an example in which the CourseMan software is configured using its domain model (developed in Chapter 3). To ease reading, the command is broken down into three lines. In practice, **you must type the whole command in one line**!

```
java -Dlogging=true -
  cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:
  memorybasedjavacompiler.jar:courseman.jar[10]
domainapp.basics.apps.tool.DomainAppTool
configure
```

10 for Windows, use the semi-colon ';' instead of the colon ':' to separate the class path elements

```
4 vn.com.courseman.model.CourseModule
  vn.com.courseman.model.CompulsoryModule
  vn.com.courseman.model.ElectiveModule
  vn.com.courseman.model.Enrolment
  vn.com.courseman.model.Student
  vn.com.courseman.model.City
  vn.com.courseman.model.SClass
```

Lines 1 and 2 specify the `java` command that runs the main class of the tool (`domainapp.basics.apps.tool.DomainAppTool`), with the class path (option: `-cp`) that refers to the library files that are provided by the tool. The first 4 files are core library files that must be included for every software. The fifth file (`courseman.jar`), is the domain model binary file of the software.

Lines 3 and 4 are the command line arguments that will be passed as input for the `main` method of the class `DomainAppTool`. Line 3 is the command named `configure`, which must be used in this case. Line 4 lists the domain classes of the software's domain model binary.

Note that the class path elements (which follow the `-cp` option) must be separated by either the colon ':' (as shown in this example) for Linux-based OS or by the semi-colon ';' for Windows-based OS.

The JVM option `-Dlogging=true` is not required but needed in order to display key progress information (such as the one shown below) on the command line interface.

When the command has finished its execution, a new sub-directory, called `data`, will be created in the current working directory. This directory contains the database of the software. In addition, if the logging option is turned on then progress information, such as the one shown below, will be printed on the terminal:

```
LOADING domain classes...
[vn.com.courseman.model.CourseModule,
vn.com.courseman.model.CompulsoryModule,
vn.com.courseman.model.ElectiveModule,
vn.com.courseman.model.Enrolment,
vn.com.courseman.model.Student,
vn.com.courseman.model.City,
vn.com.courseman.model.SClass]
Running setting up command: Configure...
Setting up the program...
Validating program settings...
Creating the configuration
Registering configuration schema
Deleting configuration data
Creating domain classes
Creating domain configuration
Running the application...
```

5.2.3. Running the software

Once the software has been configured, it can be run with just the domain model, without a command. For example, the following example (all in one line) runs the CourseMan software with all the domain classes in the model:

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
piler.jar:courseman.jar domainapp.basics.apps.tool.DomainAppTool
vn.com.courseman.model.CourseModule
vn.com.courseman.model.CompulsoryModule
vn.com.courseman.model.ElectiveModule
vn.com.courseman.model.Enrolment
vn.com.courseman.model.Student
vn.com.courseman.model.City
vn.com.courseman.model.SClass
```

Using a sub-set of domain classes

Note that the software can be run with a sub-set of the domain classes in the model. This is useful for cases where only a sub-set of these classes need to be tested by the user. For example, the following command will run the default software with just `Student` and `City`.

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:
```

```
memorybasedjavacompiler.jar:courseman.jar
domainapp.basics.apps.tool.DomainAppTool
vn.com.courseman.model.Student
vn.com.courseman.model.City
```

5.2.4. Deleting domain data

This is the first of three commands that are used for managing the software. This command, `deletedomaindata`, is used to delete the existing data objects of certain the domain classes in the underlying database. It, however, does not remove the data stores of the domain classes themselves. This command is thus useful if you want to reset the data stores of the classes to the initial state.

For example, to delete all the data objects of `Student` and `City`, issue this command:

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
piler.jar:courseman.jar domainapp.basics.apps.tool.DomainAppTool
deletedomaindata vn.com.courseman.model.Student
vn.com.courseman.model.City
```

Note that due to association constraints that may exist between the domain classes, deleting from one data store may fail with the exception `DataSourceException.FAIL_TO_UPDATE_OBJECT_BY_QUERY`. Thus, it is necessary to check the associations between the domain classes when deciding which data should be removed.

5.2.5. Deleting domain schema

The next software management command is `deletedomainschema`, which not only deletes all the data objects but also the data stores of one or more domain classes from the underlying database. This command is used when design changes of a domain class necessitate to regenerate the data store of that domain class in the database.

For example, to delete the domain schema concerning the domain class `Enrolment`, issue this command:

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
```

```
piler.jar:courseman.jar domainapp.basics.apps.tool.DomainAppTool
deletedomainschema vn.com.courseman.model.Enrolment
```

Note that deleting one data store may result in a cascade deletion of data in other related data stores. Thus, ensure to check the associations between the domain classes when deciding which data stores should be removed. If in doubt, re-configure the software as explained in Section 5.2.2.

5.2.6. Creating the domain schema

To recreate the domain schema after it has been deleted, use the third software management command named `createdomainschema`. Note a key difference between `createdomainschema` and `configure` (which both change the database content) that the former operates on the domain schema of the database, while the latter operates on the Config schema.

The following example shows how the command is used to recreate the domain schema of all the domain classes of the CourseMan model:

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
piler.jar:courseman.jar domainapp.basics.apps.tool.DomainAppTool
createdomainschema vn.com.courseman.model.CourseModule
vn.com.courseman.model.CompulsoryModule
vn.com.courseman.model.ElectiveModule
vn.com.courseman.model.Enrolment
vn.com.courseman.model.Student
vn.com.courseman.model.City
vn.com.courseman.model.SClass
```

5.2.7. Deleting the configuration

When the domain model has been changed in such a way that affect the object forms of the domain classes or when the tool itself has been upgraded, the Config schema of the software needs to be reinitialised. To delete this schema, use the command `deleteconfig`. Note that this command does not require the domain classes as input, and thus the class path may exclude the domain model binary.

The following example illustrates how this command is used to delete the Config schema of CourseMan software:

```
java -Dlogging=true -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
piler.jar domainapp.basics.apps.tool.DomainAppTool deleteconfig
```

5.2.8. Simulating the software

During the development phase, when the software undergoes constant changes (in most cases, due to changes to the user requirements) and these changes need to be observed and confirmed before being committed to the software configuration, we can shorten the configure-run cycle described in Sections 5.2.2 and 5.2.3 by 'simulating' the software such that its configuration is created only in memory and immediately used to run the software. This can be achieved by running the software (as explained in in Section 5.2.3) with this JVM option: `-Ddomainapp.setup.SerialiseConfiguration=false`.

For example, the following command illustrates how to simulate the CourseMan software:

```
java -Dlogging=true -Ddomainapp.setup.SerialiseConfiguration=false -
cp .:domainapptool.jar:derby.jar:scrollabledesktop.jar:memorybasedjavacom
piler.jar:courseman.jar domainapp.basics.apps.tool.DomainAppTool
vn.com.courseman.model.CourseModule
vn.com.courseman.model.CompulsoryModule
vn.com.courseman.model.ElectiveModule
vn.com.courseman.model.Enrolment
vn.com.courseman.model.Student
vn.com.courseman.model.City
vn.com.courseman.model.SClass
```

Note the following key points about this command:

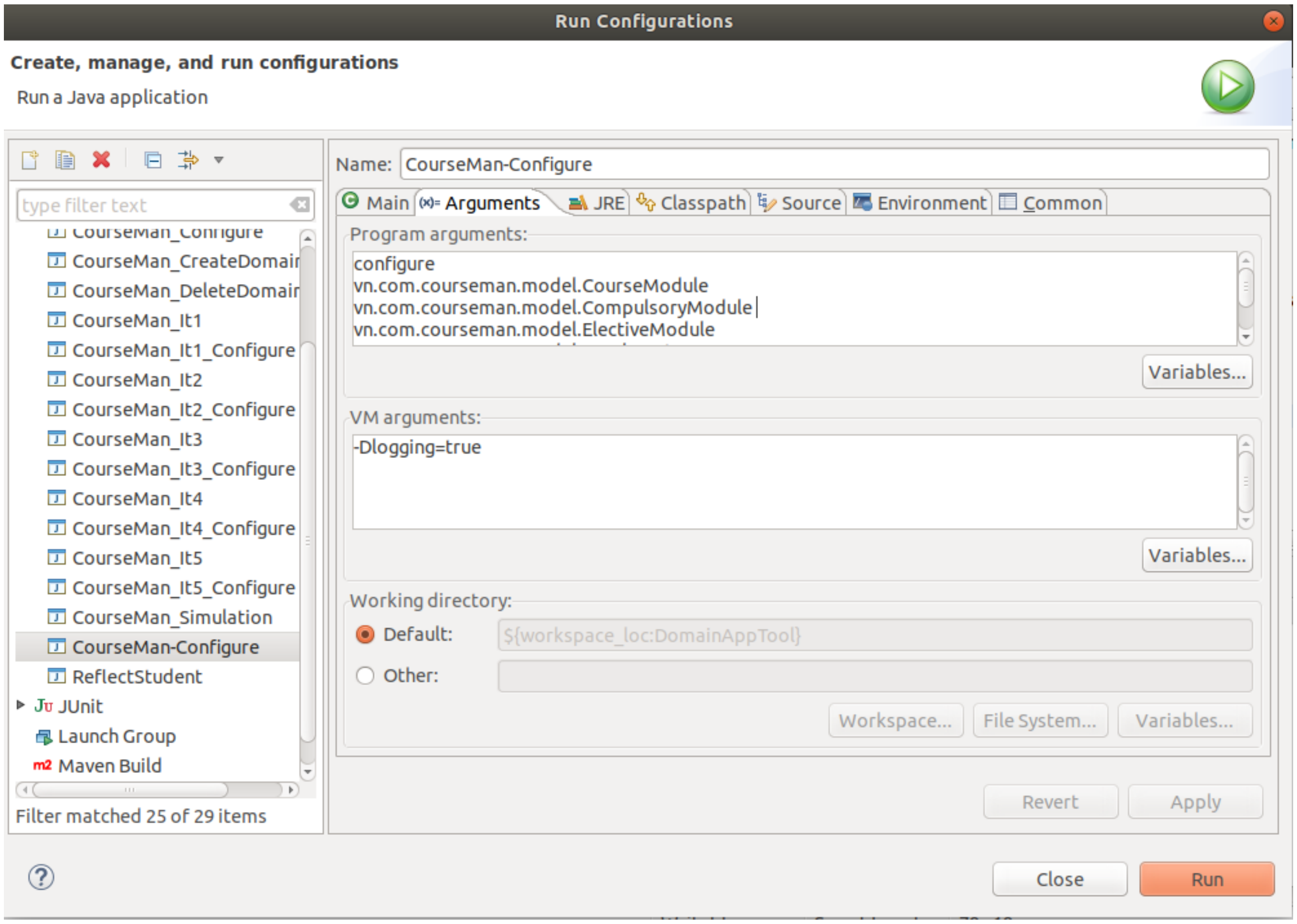
- the domain schema of the software is still created and populated in the database. Thus, any domain objects that are created/updated by the simulated software are also available in the subsequent normal (non-simulated) runs of the software.
- the existing software configuration (if any) is removed. Thus, the software needs to be reconfigured before any subsequent normal runs can take place.
- Do not use the JVM option (`domainapp.setup.SerialiseConfiguration`) with other software commands. Only use it with the run command.

5.3. Using an IDE

The software commands explained in Section 5.2 can be executed within an Integrated Development Environment (IDE), such as Eclipse and Netbeans. This alternative command-line execution method is particularly suited for working with the software during the development phase, when we want to quickly test and debug the software. The method described in Section 5.2 is used for running the software in the production environment.

The basic idea is to create in the IDE a Java project for the software and, within this project, a *run configuration* for each software command. The main class to use for all these run configurations is this class: `domainapp.basics.apps.tool.DomainAppTool`. The run configurations only differ in the program arguments and the JVM options to use. These elements are as specified in each subsection of Section 5.2 that corresponds to a command.

For example, Figure 15 illustrates how to create in an Eclipse project for CourseMan a run configuration for the software command named `configure`. The “Program arguments” include the command name and the domain class list. The “VM argument” includes the option `-Dlogging=true`. The class path is specified for the project and thus needs not be specified for each run configuration.


Name: CourseMan-Configure

Program arguments:

```
configure
vn.com.courseman.model.CourseModule
vn.com.courseman.model.CompulsoryModule
vn.com.courseman.model.ElectiveModule
```

VM arguments:

```
-Dlogging=true
```

Working directory: Default: ${workspace_loc:DomainAppTool}

Figure 15: An Eclipse's run configuration for the COURSEMAN's `configure` command.

5.4. Using a software generation API

Instead of using the command-line interface, we can create a **software class**, that encapsulates all the program arguments needed to run the software commands, and execute this software class. Designing this class is very straight-forward. Below are three simple design rules:

- a sub-class of a class named `DomainAppToolSoftware`
- overrides the method `getModel` to return an array of the domain classes
- has a `main` method whose body simply invokes the method `exec` of `DomainAppToolSoftware`, passing the entire `args` array as input.

For example, Listing 21 below shows the code of a software class for COURSEMAN named `CourseManSoftware`. The source code of this class is provided in the package named `vn.com.courseman.software` of the source code attached to this book.

Listing 21: The software class `CourseManSoftware` for running CourseMan.

```java
package vn.com.courseman.software;

import vn.com.courseman.model.City;
import vn.com.courseman.model.CompulsoryModule;
import vn.com.courseman.model.CourseModule;
import vn.com.courseman.model.ElectiveModule;
import vn.com.courseman.model.Enrolment;
import vn.com.courseman.model.SClass;
import vn.com.courseman.model.Student;
import domainapp.basics.exceptions.NotPossibleException;
import domainapp.basics.software.DomainAppToolSoftware;

/**
 * @overview
 *  Encapsulate the basic functions for setting up and running a software given its
domain model.
 *
 * @author dmle
 */
public class CourseManSoftware extends DomainAppToolSoftware {
  // the domain model of software
  private static final Class[] model = {
      CourseModule.class,
      CompulsoryModule.class,
      ElectiveModule.class,
      Enrolment.class,
      Student.class,
      City.class,
      SClass.class
  };
  /**
   * @effects return {@link #model}.
   */
  @Override
  protected Class[] getModel() { return model; }
  /**
   * The main method
   * @effects
   *  run software with a command specified in args[0] and with the model
   *  specified by {@link #getModel()}.
   *
   *  <br>Throws NotPossibleException if failed for some reasons.
   */
  public static void main(String[] args) throws NotPossibleException {
    new CourseManSoftware().exec(args);
  }
}
```

Note that, for performance reasons, method `getModel` should not create and return the domain class array directly. Instead, it should return the value of a static variable, which is initialised to this array. In Listing 21, for instance, method `CourseManSoftware.getModel` returns the value of the static variable named `model`. This variable is initialised to the domain class array of the CourseMan's domain model. When need to, this variable can be customised to hold a sub-set of the domain classes of the model.

Exercises

Ex1. Name two ways in which a software is generated using DomainAppTool. Briefly discuss the pros and cons of each.
Ex2. Describe the structure of the underlying database of a software.
Ex3. What are the software tasks supported by DomainAppTool.
Ex4. What is the DomainAppTool command that is used for each software task?
Ex5. What are the types of binary that you need in order to execute the software tasks in DomainAppTool?
Ex6. Briefly discuss the types of interface that DomainAppTool provides the developers for performing the software tasks.
Ex7. What is the main executable class that is used for performing the software tasks via the command-line interface?
Ex8. What are two types of command line arguments necessary to execute each software task via the command-line interface?

Ex9. Which software task requires the developer to specify an additional JVM option when executing? Briefly explain this option.

Ex10. What is the main difference between executing the software tasks using the command-line interface of the OS and executing these tasks using an IDE (e.g. Eclipse)?

Chapter 6. Developing reports

A report type is a domain class (as per the design in Chapter 3) which represents a view over the state(s) of domain objects of other non-report, yet, related domain classes. We call the domain class that models a report type **report class** and the non-report domain classes involved the **input domain classes**. The view that the report class represents is called a **report query**.

A report class differs from the non-report ones in that its domain attributes are generally organised into two logical groups: input and output attributes. **Input attributes**, which may or may not be present, model the user input parameters for the report query. In contrast, **output attributes** are those that are computed from the report query's result.

We will explain the design rules for the report class and its structure shortly. But first, let us discuss the example student-by-name report of the COURSEMAN software.

Example: Students by name report

The requirement of this report was introduced in Section 1.2. Technically, this report provides a view over the state of the `Student` objects, the values of whose `name` attribute contains a user's specified name value.

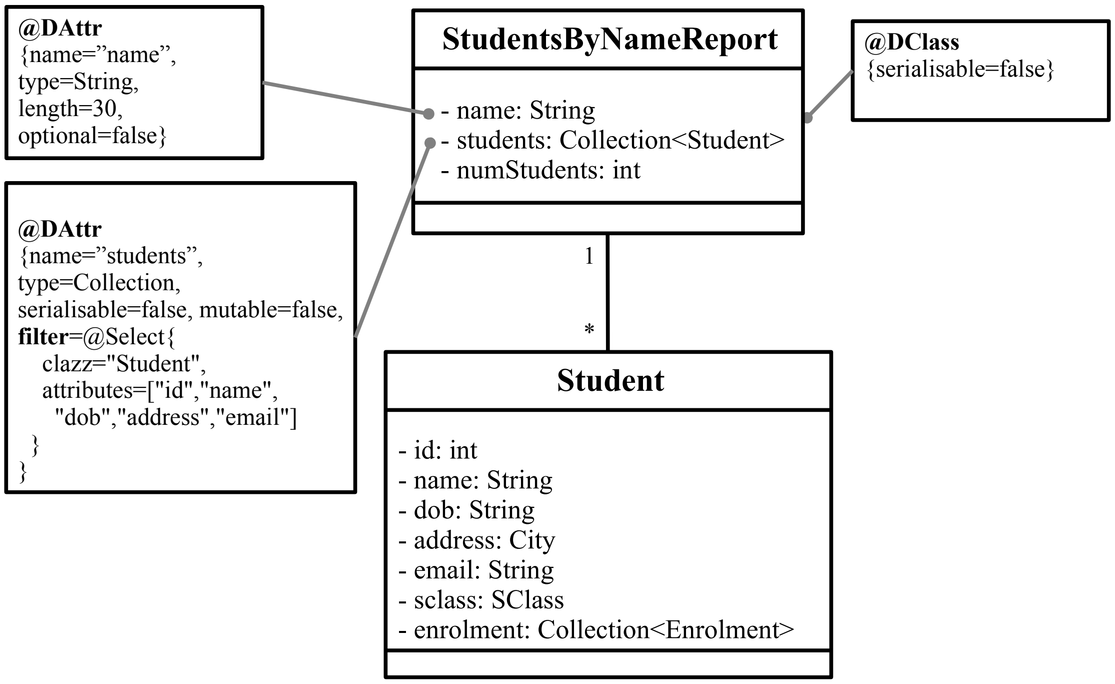
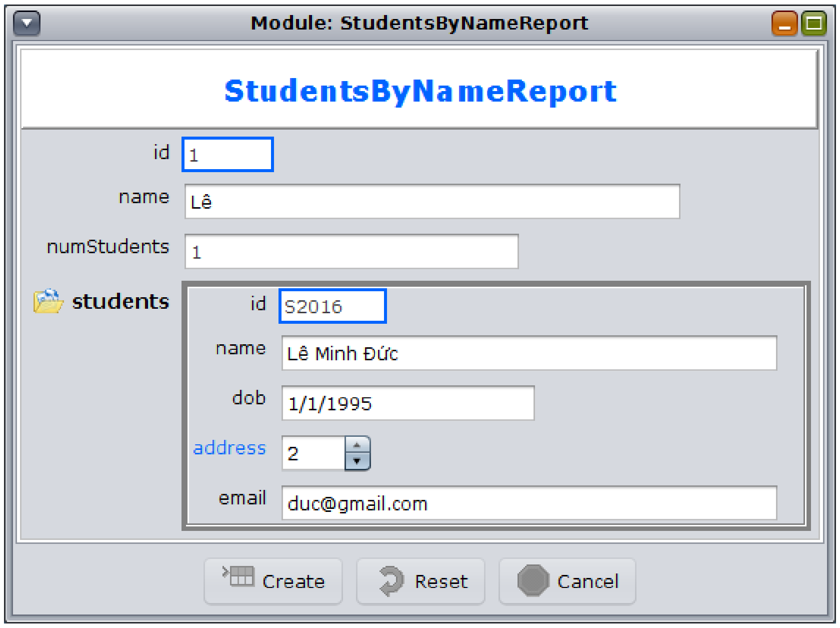


Figure 16: Report class `StudentsByNameReport` as a domain class.

The report query, written in the Structured Query Language (SQL) [31], is as follows:

```
Select * from student Where name like '%Le%'
```

Figure 16 shows the design of the report class named `StudentsByNameReport`. This class has three domain attributes: `name`, `students` and `numStudents`. Attribute `name` is the input attribute and models the input parameter of the report. It has the same design specification as the attribute `Student.name`. Attribute `students` and `numStudents` are output attributes. In particular, attribute `students` is an asociative attribute that realises the report's end of the association to `Student`. This attribute holds the result of the report query.



id	1
name	Lê
numStudents	1

id	S2016
name	Lê Minh Đức
dob	1/1/1995
address	2
email	duc@gmail.com

Figure 17: The generated report form of `StudentByNameReport`.

Figure 17 shows the **report form** of `StudentByNameReport` that is generated by DomainAppTool. It is the object form in the `StudentByNameReport`'s view. We will discuss this form in more detail later in Section 6.5. What is important to understand here is that the report form has the same design as the object form and, thus, requires very little extra effort to understand. As will be explained later, the extra effort only involves understanding the logical input and output regions of the form.

6.1. Report class

Typically, we would design this class as non-serialisable, because its objects are not to be stored in the data source. This is achieved by setting `DClass.serialisable=false`.

Example: StudentsByNameReport

Figure 16 shows how the report class `StudentsByNameReport` is configured as a non-serialisable domain class. Listing 22 shows the Java code of the header of this class.

Listing 22: Report class `StudentsByNameReport`

```java
/**
 * @overview
 *   Represent the reports about students by name.
 * @author dmle
 */
@DClass(serialisable=false)
public class StudentsByNameReport { }
```

6.2. Input attribute

Input attributes are mutable domain attributes (configured with `DAttr.mutable=true`). These attributes, which may or may not be present, model the user input parameters of the report query. Each input attribute matches the domain attribute of an input domain class.

Example: StudentsByNameReport

Listing 23 shows how the input attribute `StudentsByNameReport.name` (shown in Figure 16) is defined in Java. In this case `Student` is the input domain class of `StudentsByNameReport`. Attribute `StudentsByNameReport.name` corresponds to the attribute `Student.name` shown in Listing 2 (its `DAttr` specification is exactly the same as that of `Student.name`).

Listing 23: The input attribute `StudentsByNameReport.name`

```java
/**
 * ...as shown in Listing 22... */
@DClass(serialisable=false)
public class StudentsByNameReport {
  /**input: student name */
  @DAttr(name = "name", type = Type.String, length = 30, optional = false)
  private String name;
}
```

6.3. Output attribute

An output attribute models either the report query's result or a value that is computed from this result. Output attributes are domain attributes that have the following specification:

- is immutable, i.e. `DAttr.mutable=false`.
- is derived from the input attributes: property `DAttr.derivedFrom` includes names of the input attributes.

  An exception to this rule is when an output attribute is itself derived from another output attribute. In this case, property `DAttr.derivedFrom` needs *not* be specified.
- is defined with the annotation `Output`. This annotation is used by the tool to read and process output attributes.

In addition to these rules, if an output attribute models an association to an input domain class then other rules concerning this association need to be applied (as described in Chapter 3).

6.3.1. Associative output attribute

If an output attribute is a collection-typed attribute and models an association to an input domain class then it records the objects of this class that satisfy the query. In this case:

- the output attribute must model a one-many association with the input domain class, in which the report class is the one end
  - property `DAttr.ref.filter` can specify names of the domain attributes of the associated domain class that appear on the output form. This list must contain the report domain attribute of the associated domain class (explained below).
- the report class is defined with one single 'dummy' link-adder operation for the output attribute (basically this operation does nothing and returns an arbitrary `boolean` value)

On the other hand, the associated input domain class needs to:

- have an associative attribute named the *report domain attribute* that
  - realises the many end of the association,
  - needs not be defined with a `DAssoc` and
  - is non-serialisable and virtual (i.e. `DAttr.serialisable=false, virtual=true`)
- have a getter operation for this associative attribute

6.3.2. Non-associative output attribute

If an output attribute is non-associative then it must be defined with `DAttr.auto=true`.

Example: StudentsByNameReport

Listing 24 shows the complete Java code of the report class `StudentsByNameReport`. In this example, we will focus on the code segments concerning the two output attributes of this class, namely `students` and `numStudents`.

Notice how both output attributes are defined with `DAttr.mutable=false` and with the annotation `Output`. Attribute `numStudents` is non-associative so it is defined with `DAttr.auto=true`. Attribute `students`, on the other hand, is associative and used to record all objects of the input domain class `Student` that satisfy the query. This explains why this attribute is defined with a `DAssoc` that models a one-many association with the input domain class `Student`. We use the property `DAttr.filter` of this attribute to specify a list of domain attributes of interest to the report. These include `id`, `name`, `dob`, `address`, `email` and the report domain attribute (`rptStudentByName`). Thus, the report users will not be able to see the enrolments of a particular student.

The link-adder operation named `addStudent` is a dummy operation that returns `false`.

Listing 25 shows changes to the domain class `Student` to support the demand of `StudentsbyNameReport`. The essential changes include the associative attribute named `rptStudentByName` and a getter operation for this attribute. In addition, we also define a number of attribute name constants (shown at the top of the listing) for use in the definition of the property `DAttr.filter` of the attribute `StudentsByNameReport.students`.

Listing 24: The complete Java code of `StudentsByNameReport`

```
package vn.com.courseman.model.reports;

import java.util.Map;
import domainapp.basics.core.dodm.dsm.DSMBasic;
import domainapp.basics.core.dodm.qrm.QRM;
import domainapp.basics.exceptions.DataSourceException;
import domainapp.basics.exceptions.NotPossibleException;
import domainapp.basics.model.Oid;
import domainapp.basics.model.query.Expression.Op;
import domainapp.basics.model.query.Query;
import domainapp.basics.model.query.QueryToolKit;
import domainapp.modules.report.model.meta.Output;
import vn.com.courseman.model.Student;
// other imports: omitted
/**
 * ...as shown in Listing 22...
 */
@DClass(schema="courseman",serialisable=false)
public class StudentsByNameReport {
   @DAttr(name = "id", id = true, auto = true, type = Type.Integer, length = 5,
optional = false, mutable = false)
  private int id;
  private static int idCounter = 0;

  /**input: student name */
  @DAttr(name = "name", type = Type.String, length = 30, optional = false)
  private String name;

  /**output: students whose names match {@link #name} */
  @DAttr(name="students",type=Type.Collection,optional=false, mutable=false,
      serialisable=false,filter=@Select(clazz=Student.class,
      attributes={Student.A_id, Student.A_name, Student.A_dob, Student.A_address,
                  Student.A_email,Student.A_rptStudentByName}),derivedFrom={"name"})
  @DAssoc(ascName="students-by-name-report-has-students",role="report",
      ascType=AssocType.One2Many,endType=AssocEndType.One,

associate=@Associate(type=Student.class,cardMin=0,cardMax=MetaConstants.CARD_MORE))
  @Output
  private Collection<Student> students;

  /**output: number of students found (if any), derived from {@link #students} */
  @DAttr(name = "numStudents", type = Type.Integer, length = 20, auto=true,
mutable=false)
```

```
  @Output
  private int numStudents;

  /**
   * @effects
   *  initialise this with <tt>name</tt> and use {@link QRM} to retrieve from data
source
   *  all {@link Student} whose names match <tt>name</tt>.
   *  initialise {@link #students} with the result if any.
   *
   *  <p>throws NotPossibleException if failed to generate data source query;
   *  DataSourceException if fails to read from the data source
   *
   */
  @DOpt(type=DOpt.Type.ObjectFormConstructor)
  @DOpt(type=DOpt.Type.RequiredConstructor)
  public StudentsByNameReport(@AttrRef("name") String name) throws
NotPossibleException, DataSourceException {
    this.id=++idCounter;
    this.name = name;
    doReportQuery();
  }

  /**
   * @effects return name
   */
  public String getName() {
    return name;
  }

  /**
   * @effects <pre>
   *  set this.name = name
   *  if name is changed
   *    invoke {@link #doReportQuery()} to update the output attribute value
   *    throws NotPossibleException if failed to generate data source query;
   *    DataSourceException if fails to read from the data source.
   *  </pre>
   */
  public void setName(String name) throws NotPossibleException, DataSourceException {
    this.name = name;

    // DONOT invoke this here if there are > 1 input attributes!
    doReportQuery();
  }

  /**
   * This method is invoked when the report input has been set by the user.
   *
   * @effects <pre>
   *   formulate the object query
   *   execute the query to retrieve from the data source the domain objects that
satisfy it
   *   update the output attributes accordingly.
   *
   *  <p>throws NotPossibleException if failed to generate data source query;
   *  DataSourceException if fails to read from the data source. </pre>
   */
  @DOpt(type=DOpt.Type.DerivedAttributeUpdater)
  @AttrRef(value="students")
  public void doReportQuery() throws NotPossibleException, DataSourceException {
    // the query manager instance
    QRM qrm = QRM.getInstance();

    // create a query to look up Student from the data source
    // and then populate the output attribute (students) with the result
    DSMBasic dsm = qrm.getDsm();

    Query q = QueryToolKit.createSearchQuery(dsm, Student.class,
        new String[] {Student.A_name},
        new Op[] {Op.MATCH},
        new Object[] {"%"+name+"%"});
    Map<Oid, Student> result = qrm.getDom().retrieveObjects(Student.class, q);


    if (result != null) {
      // update the main output data
      students = result.values();
      // update other output (if any)
      numStudents = students.size();
    } else {
      // no data found: reset output
      resetOutput();
    }
  }

  /**
   * @effects
   *  reset all output attributes to their initial values
   */
  private void resetOutput() {
    students = null;
    numStudents = 0;
  }

  /**
   * A link-adder method for {@link #students}, required for the object form to
function.
   * However, this method is empty because students have already be recorded in the
attribute {@link #students}.
   */
  @DOpt(type=DOpt.Type.LinkAdder)
  public boolean addStudent(Collection<Student> students) {
    // do nothing
    return false;
  }
```

```
/**
 * @effects return students
 */
public Collection<Student> getStudents() {
  return students; }

/**
 * @effects return numStudents
 */
public int getNumStudents() {
  return numStudents;
}

/**
 * @effects return id
 */
public int getId() {
  return id;
}

@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + id;
  return result;
}

@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
    StudentsByNameReport other = (StudentsByNameReport) obj;
    if (id != other.id)
      return false;
    return true;
  }

  @Override
  public String toString() {
    return "StudentsByNameReport (" + id + ", " + name + ")";
  }
}
```

Listing 25: Changes to the `Student` class to support `StudentsByNameReport`

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

  // attribute name constants (omitted)
  public static final String A_name = "name";
  public static final String A_id = "id";
  public static final String A_dob = "dob";
  public static final String A_address = "address";
  public static final String A_email = "email";
  public static final String A_rptStudentByName = "rptStudentByName";

  // domain attributes (omitted)

  // realise link to report
  @DAttr(name=A_rptStudentByName,type=Type.Domain, serialisable=false,
      // IMPORTANT: set virtual=true to exclude this attribute from the object state
      // (avoiding the view having to load this attribute's value from data source)
      virtual=true)
  private StudentsByNameReport rptStudentByName;

  /**
   * @effects return rptStudentByName
   */
  public StudentsByNameReport getRptStudentByName() {
    return rptStudentByName;
  }
  // other code (omitted)
}
```

6.4. Report query

Writing the report query is a key task in report class design. In its most basic form, a report query is an object query over the input domain classes. It seeks to retrieve from the data source a set of domain objects of this class that satisfy a set of criteria. These criteria are expressions over the domain attributes of the input domain classes. These expressions may have variables that are set by values of input attributes.

The query is written and executed using an **object query** API provided by the tool. This API provides two classes: `QueryToolKit` and `QRM` (abbrv. Query Manager). Class `QueryToolKit` is used to generate the query from its parts. Class `QRM` then provides access to a `DOMBasic` object (provided by the object manager component discussed in Section 2.4). This object is used to execute the query.

The object query whose result is to be set to an associative output attribute takes the following general form:

Find $\{o \mid o : c,\ c_1.a_1\ op_1\ v_1, \ldots,\ c_n.a_n\ op_n\ v_n\}$.

where:

- $o : c$ denotes a domain object o of a domain class c
- $c_i.a_i\ op_i\ v_i\ (i = 1 \ldots n)$ are query expressions
- $c_1,\ldots,c_n$ are the input domain classes; $c \in \{c_1,\ldots,c_n\}$
- $a_1,\ldots,a_n$ are the input attributes
- $op_1,\ldots,op_n$ are query operators (defined by the enum `Op`)
- $v_1,\ldots,v_n$ are values of the input attributes

In this book, we will assume that non-associative output attributes are also computed from the above query.

6.4.1. Query method

In the report class, the report query is defined in a method so that it can be executed easily when needed to. We call this method **query method** and conveniently name it `doReportQuery`. The header of this method must have the following parts (as shown in Listing 24):

- `@DOpt(type=DOpt.Type.DerivedAttributeUpdater)`
- `@AttrRef` whose `value` property is set to name of the main output attribute (usually the associative one)
- access modifier `public`
- throws two exceptions: `NotPossibleException`, `DataSourceException`

Behaviourally, the query method formulates the object query, executes it and use the result (`null` or not) to update the output attribute(s). Note the followings about the invocation points of this method:

- query method is invoked by the constructor of the report class
- if the report class has only one input attribute then the query method is invoked by the setter operation of this attribute. Otherwise, it needs not be invoked by the setter methods, because it will be invoked automatically by DomainAppTool.

The next example will explain the DomainAppTool's API for implementing the behaviour of the query method.

Example: StudentsByNameReport

Listing 24 shows how the object query of the report class `StudentsByNameReport` is defined and executed and how the result is used to update the two output attributes. The query method is `doReportQuery`. This method is invoked by both the constructor and method `setName`. The latter invocation is because `StudentsByNameReport` has only one input attribute.

The following points should be noted about the method body. First, the object query is formulated by invoking the method `QueryToolKit.createObjectQuery` and passing in the query parts as input. These include an instance of the `DSMBasic` object (obtained from the `QRM`), the input domain class (`Student` in this case), and three equal-sized arrays: the first contains the input domain attribute names, second contains the operators, and the third contains the values. The object query created in Listing 24 is the following query:

```
Find {s | s : Student, Student.name ≃ "%name%" }.
```

The literal `"%name%"` is a string pattern that has the similar form to the SQL value pattern. The variable `name` used in this pattern is the value of the input attribute `name`. This query is equivalent to the report's SQL query given in the example at the beginning of this section.

Second, to execute the object query requires invoking the method `DOMBasic.retrieveObjects` and passing in the input domain class (`Student` in this case) and the object query. The result is a `Map` object that maps the object IDs (type: `OID`) of the objects that satisfy the query.

Finally, this result is used to update the output attributes. Note that these attributes are reset if result is `null` (i.e. no objects were found).

6.5. Working with the report GUI

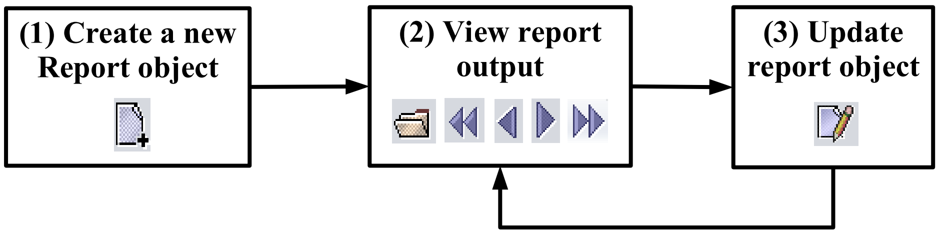


Figure 18: A standard procedure for working with the report GUI.

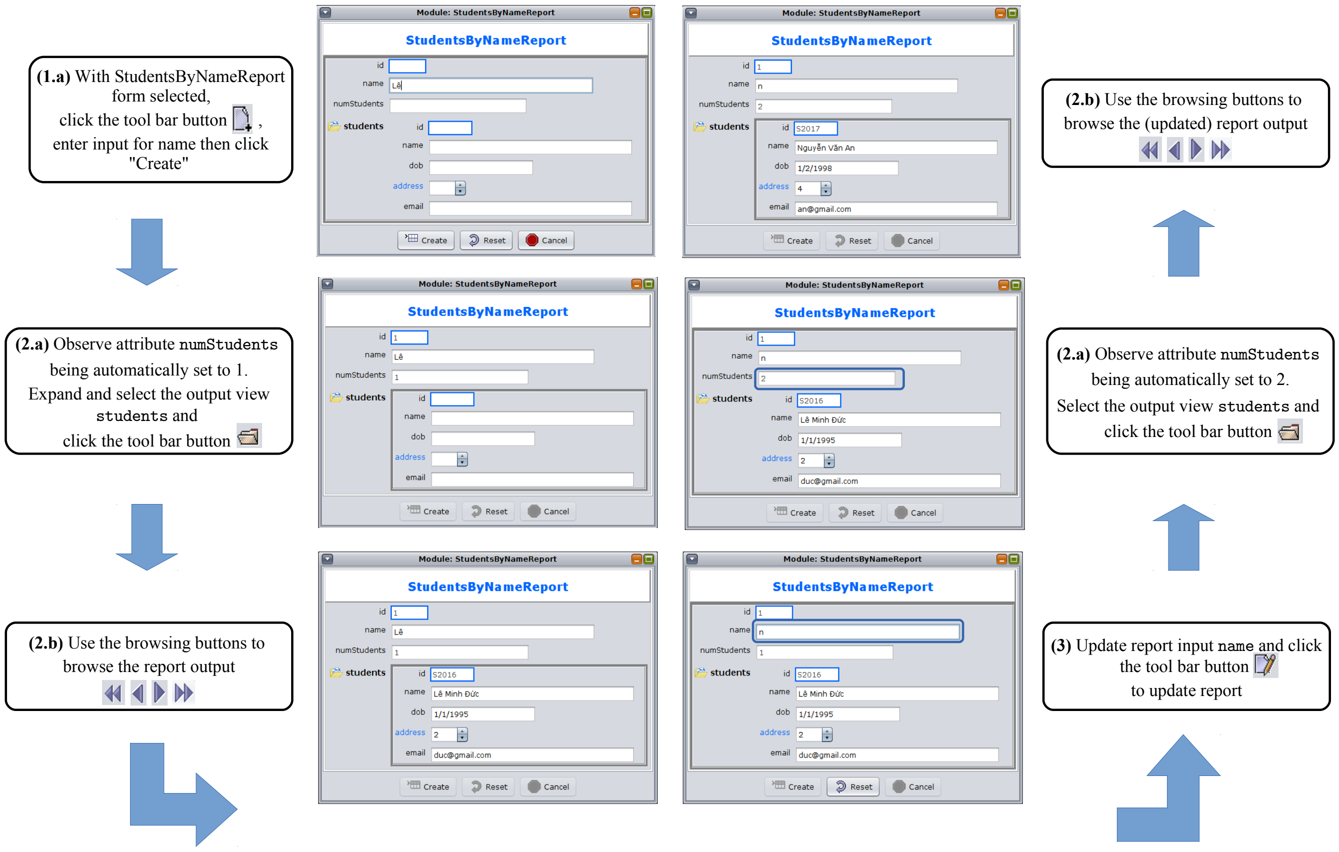

(1.a) With StudentsByNameReport form selected, click the tool bar button , enter input for name then click "Create"
(2.a) Observe attribute numStudents being automatically set to 1. Expand and select the output view students and click the tool bar button
(2.b) Use the browsing buttons to browse the report output
(3) Update report input name and click the tool bar button to update report
(2.a) Observe attribute numStudents being automatically set to 2. Select the output view students and click the tool bar button
(2.b) Use the browsing buttons to browse the (updated) report output
Module: StudentsByNameReport
StudentsByNameReport
id
name
numStudents
students
dob
address
email
Create
Reset
Cancel

Figure 19: Applying the 3-step procedure to the report GUI of `StudentsByNameReport`.

The report GUI consists of two main regions: (1) an **input region** for displaying the input attributes and (2) an **output region** for presenting the output ones. The input region contains data fields that are used to obtain user input for the input attributes. The output region contains data fields (for non-associative output attributes) and sub-forms (for associative output attributes). For example, the input region of the report form shown in Figure 17 consists of two data fields: `id` and `name`; while the output region of this form consists of one data field (`numStudents`) and one sub-form (`students`).

In addition to following the general GUI guidelines explained in Chapter 2.5, the user should observe a standard procedure for working with the report GUI. This procedure is shown in Figure 18 and consists of 3 basic steps. Each step is illustrated with the tool bar buttons that are used. Note that steps 2 and 3 are performed repeatedly until the user is satisfied with the report output.

Figure 19 illustrates this procedure using the `StudentsByNameReport` example. The block arrows show the detailed step flow, in which step 2 is broken down into two smaller steps (2a and 2b). Step 2a instructs the user on how to open the result sub-form, while step 2b explains that the browsing buttons be used to navigate the result objects that are displayed on this form.

6.6. Report with join queries

In practice, reports often involves executing a query over two or more associated domain classes. At run-time, this query is translated into an SQL join query for execution. To illustrate, let us consider a report that seeks to answer the following question:

Find all students whose address are any city whose name has the prefix 'H'.

We can express this query precisely as the following object query:

```
Find {o | o : Student,
      Student join City on Student.address, City.name matches 'H%'} }
```

This query is translated into the following SQL join query[11]:

```
Select * from Student as t1, City as t2 where t1.address = t2.id and
                          t2.name like 'H%'
```

The report class for the above query is called `StudentsByCityJoinReport`, whose design is shown in Figure 20. It looks very similar to the `StudentsByNameReport` class, except for the following features:

- input attribute is `cityName`, which obtains input value pattern for the `City.name` attribute
- method `doReportQuery`: uses the `createSimpleJoinQuery` method of `QueryToolKit` to generate the join query

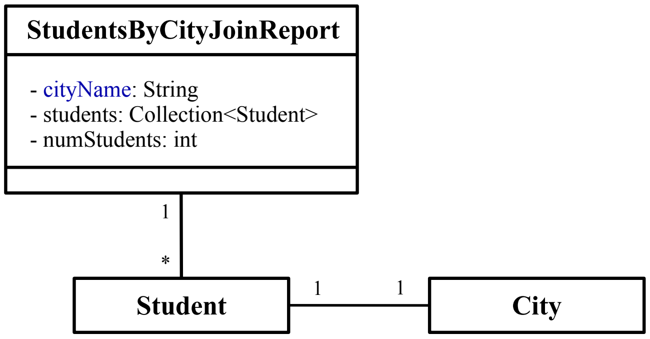


Figure 20: Design of the report class `StudentsByCityJoinReport`

11 this query can be expressed using the SQL's `join` operator.

Listing 26 shows the Java code of `StudentsByCityJoinReport`. The different elements compared to Listing 24 are highlighted. Notice how property `DAttr.filter.attributes` of the the attribute `students` includes a new associative field, named `rptStudentByCity`, which is defined in the `Student` class. This field realises `Student`'s role in the association to the report class.

Listing 26: Java code of `StudentByCityJoinReport`

```java
/**
 * @overview
 *     Represent a report about students by city whose view is expressed by a join query.
 *
 * @author ducmle
 *
 * @version 5.3
 */
@DClass(schema="courseman",serialisable=false)
public class StudentsByCityJoinReport {
  @DAttr(name = "id", id = true, auto = true, type = Type.Integer, length = 5, optional =
false, mutable = false)
  private int id;
  private static int idCounter = 0;

  /**input: city name */
  @DAttr(name = "cityName", type = Type.String, length = 30, optional = false)
  private String cityName;

  /**output: students whose names match {@link #cityName} */
  @DAttr(name="students",type=Type.Collection,optional=false, mutable=false,
      serialisable=false,filter=@Select(clazz=Student.class,
      attributes={Student.A_id, Student.A_name, Student.A_dob, Student.A_address,
          Student.A_email, Student.A_rptStudentByCity})
      ,derivedFrom={"cityName"}
      )
  @DAssoc(ascName="students-by-cityName-report-has-students",role="report",
      ascType=AssocType.One2Many,endType=AssocEndType.One,
    associate=@Associate(type=Student.class,cardMin=0,cardMax=MetaConstants.CARD_MORE
    ))
  @Output
  private Collection<Student> students;

  /**output: number of students found (if any), derived from {@link #students} */
  @DAttr(name = "numStudents", type = Type.Integer, length = 20, auto=true, mutable=false)
  @Output
  private int numStudents;
  /**
   * @effects
   *  initialise this with <tt>cityName</tt> and use {@link QRM} to retrieve from data source
   *  all {@link Student} whose addresses match {@link City}, whose names match
<tt>cityName</tt>.
   *  initialise {@link #students} with the result if any.
   *
   *  <p>throws NotPossibleException if failed to generate data source query;
   *  DataSourceException if fails to read from the data source
   *
   */
  @DOpt(type=DOpt.Type.ObjectFormConstructor)
  @DOpt(type=DOpt.Type.RequiredConstructor)
  public StudentsByCityJoinReport(@AttrRef("cityName") String name) throws
NotPossibleException, DataSourceException {
    this.id=++idCounter;

    this.cityName = name;

    doReportQuery();
  }

  /**
   * @effects return cityName
   */
  public String getCityName() {
    return cityName;
  }

  /**
   * @effects <pre>
   *  set this.name = cityName
   *  if cityName is changed
   *    invoke {@link #doReportQuery()} to update the output attribute value
   *    throws NotPossibleException if failed to generate data source query;
   *    DataSourceException if fails to read from the data source.
   *  </pre>
   */
  public void setCityName(String name) throws NotPossibleException, DataSourceException {
    this.cityName = name;

    // DONOT invoke this here if there are > 1 input attributes!
    doReportQuery();
  }

  /**
   * This method is invoked when the report input has be set by the user.
```

```
 *
 * @effects <pre>
 *   formulate the object query
 *   execute the query to retrieve from the data source the domain objects that satisfy it
 *   update the output attributes accordingly.
 *
 *  <p>throws NotPossibleException if failed to generate data source query;
 *  DataSourceException if fails to read from the data source. </pre>
 */
@DOpt(type=DOpt.Type.DerivedAttributeUpdater)
@AttrRef("students")
public void doReportQuery() throws NotPossibleException, DataSourceException {
  // the query manager instance

  QRM qrm = QRM.getInstance();

  // create a query to look up Student from the data source
  // and then populate the output attribute (students) with the result
  DSMBasic dsm = qrm.getDsm();

  ////TODO: create a 2-way join query
  Query q = QueryToolKit.createSimpleJoinQuery(dsm, Student.class, City.class,
      Student.A_address,
      City.A_name,
      Op.MATCH,
      "%"+cityName+"%");

  Map<Oid, Student> result = qrm.getDom().retrieveObjects(Student.class, q);

  if (result != null) {
    // update the main output data
    students = result.values();

    // update other output (if any)
    numStudents = students.size();
  } else {
    // no data found: reset output
    resetOutput();
  }
}

/**
 * @effects
 *  reset all output attributes to their initial values
 */
private void resetOutput() {
  students = null;
  numStudents = 0;
  }

  /**
   * A link-adder method for {@link #students}, required for the object form to function.
   * However, this method is empty because students have already be recorded in the attribute
{@link #students}.
   */
  @DOpt(type=DOpt.Type.LinkAdder)
  public boolean addStudent(Collection<Student> students) {
    // do nothing
    return false;
  }

  /**
   * @effects return students
   */
  public Collection<Student> getStudents() {
    return students;
  }

  /**
   * @effects return numStudents
   */
  public int getNumStudents() {
    return numStudents;
  }

  /**
   * @effects return id
   */
  public int getId() {
    return id;
  }

  /* (non-Javadoc)
   * @see java.lang.Object#hashCode()
   */
  /**
   * @effects
   *
   * @version
   */
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    StudentsByCityJoinReport other = (StudentsByCityJoinReport) obj;
    if (id != other.id)
      return false;
    return true;
  }

  @Override
  public String toString() {
    return this.getClass().getSimpleName()+ " (" + id + ", " + cityName + ")";
  }
}
```

6.7. Exercises

Ex1. Update the report class `StudentsByNameReport` in the example of the chapter so that it finds all the `Student` objects whose names start with the input `name`.

Ex2. Update the report class `StudentsByNameReport` in the example of the chapter so that it finds all the `Student` objects whose names either start with or end with the input `name`.

Hint: you would need to formulate and execute two report queries and then merge their results.

Ex3. Update the report class `StudentsByNameReport` in the example of the chapter so that it additionally displays the attribute `Student.enrolments` on the output.

Ex4. Create a new report class named `StudentProfilesReport` which is similar in design to `StudentsByNameReport` but contains an additional input attribute for the student year of birth. Note that the user is required to enter at least one input but is not required to enter all the input. Further, the report query must be formulated such that it only uses the input attributes that are specified.

Chapter 7. Software development method

In this section, we discuss the software development method that the development team of a software project should follow if they were to adopt the DomainAppTool for their project. We call this method **domain-driven software development method** (**DDSDM**).

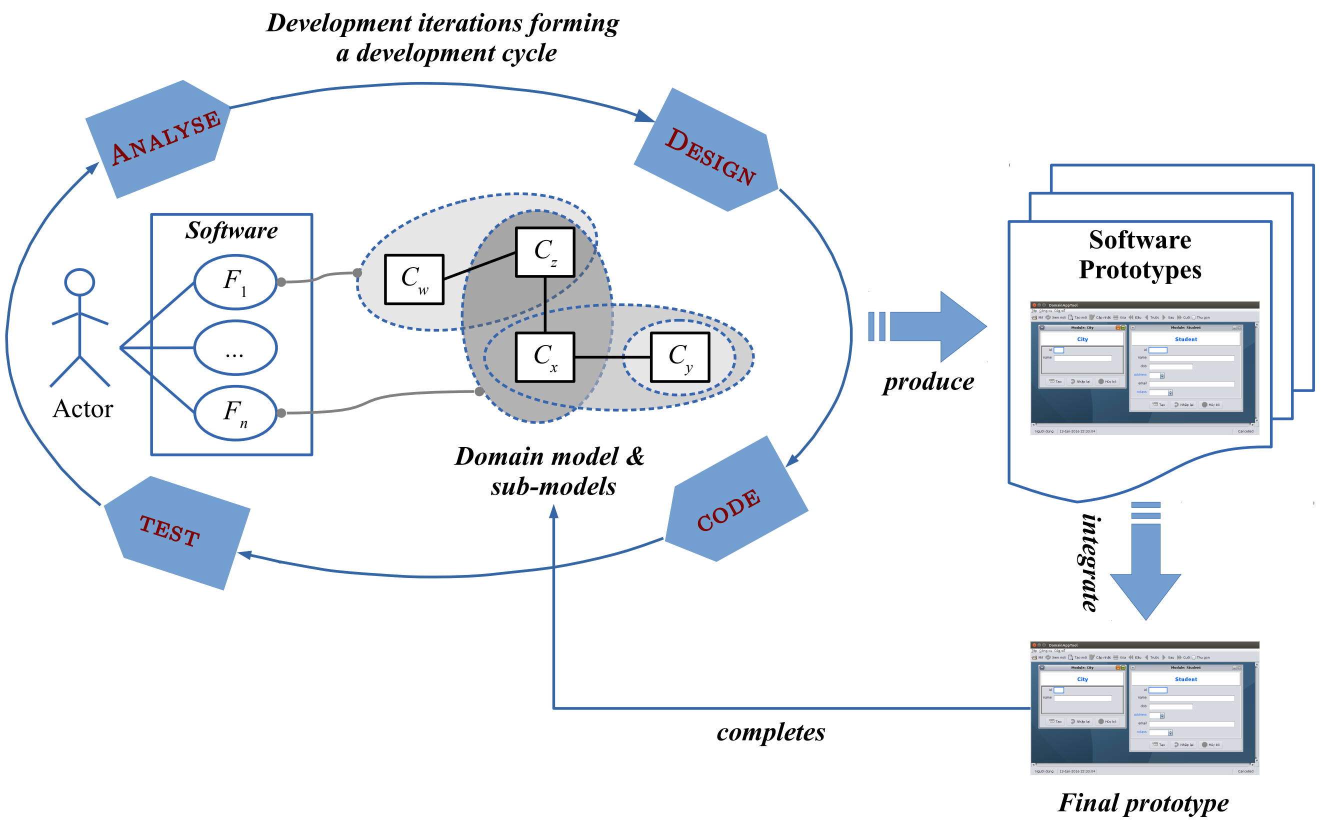


Figure 21: An overview of DDSDM.

In a nutshell, DDSDM is an iterative development method [22], [23] for developing the reusable software prototypes directly from the domain model. These prototypes are used in two ways. The first and primary usage is for the domain experts and the development team to incrementally, collaboratively and interactively develop the domain model. The second usage is to be reused in a later stage to develop the production software. In this book, we will focus on the first usage.

Figure 21 depicts DDSDM as consisting of the following phases:

1. Develop a conceptual domain model
2. Define the development iterations
3. Perform the development iterations to develop a set of software prototypes
4. Integrate the software prototypes to produce the final prototype

In the remainder of this chapter, we will discuss in detail each of these phases and illustrate them using the COURSEMAN software example.

7.1. Develop a conceptual domain model

This is a **high-level domain model** that is used as the starting point for the development process. We will use this model to define the development iterations, the performance of which will gradually enrich the domain model with new detailed features.

The high-level domain model consists of only the core (partially completed) domain classes and the initial associations among these classes. These classes and associations are identified from the functional requirements of the software. These requirements are typically described in the form of use cases [22], [23]. Each use case helps identify a subset of related domain classes and their associations in the model. We call these a **sub-model** of the domain model.

Figure 21 depicts the functional requirements using a use case model. Each use case is connected to a sub-model of the domain model. The boundary of each sub-model is represented by a dashed, filled oval, which contains one or more domain classes together with the associations among them (if any). For instance, use case F_1 is connected to a sub-model containing two domain classes, namely C_z and C_w, and an association between them.

The sub-models of two functions overlap in terms of either a shared domain class and/or an association between two domain classes of the two sub-models. It is through these overlapping points that the sub-models are combined to form the entire domain model. For example, Figure 21 shows how the aforementioned F_1's sub-model overlaps with a sub-model containing the two classes C_z, C_x in the class C_z. This sub-model in turn overlaps with another sub-model containing just the class C_y in the association between C_x and C_y.

We emphasise that our method maps uses cases to sub-models, thereby allowing the application of the construction technique described in Chapters 2-6 (for the domain model) to be used to develop the software functionality. Conceptually, this functionality is collectively defined by behaviours of the domain classes in the domain model.

Example: CourseMan

The right-hand-side of Figure 22 shows the conceptual domain model of CourseMan. This model is composed from five sub-models, which are marked in the figure by the five dashed, filled ovals. Each sub-model is connected to a use case that is shown on the left-hand-side of the figure. The first two sub-models contain the domain classes `Student` and `CourseModule`, respectively. The third sub-model contains `Student`, `CourseModule` and an association between these two classes. This association represents student enrolments into course modules. The fourth sub-model contains `SClass`, `Student` and an association between these two classes. This association represents groupings of students into classes. The fifth sub-model contains `StudentByNameReport`, `Student` and an association between these two classes. This association represents the students that appear in the result of each report.

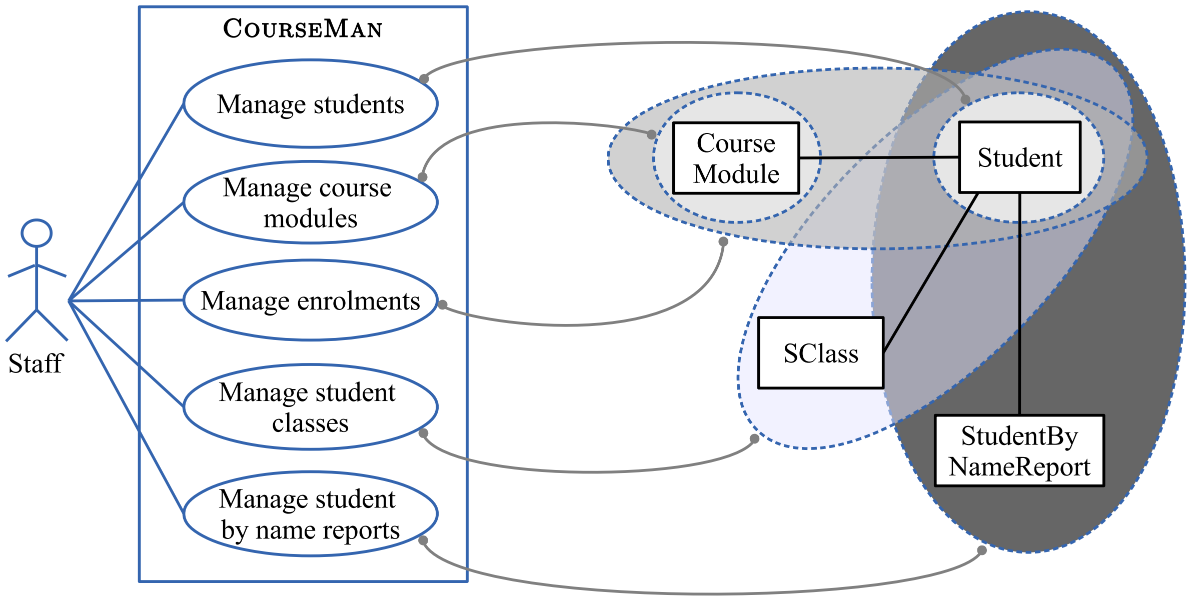


Figure 22: The conceptual domain model and sub-models of CourseMan.

7.2. Define the development iterations

Once a high-level domain model has been created, the next phase is to define the development iterations. Together, these iterations form the development plan for the software. Each iteration involves performing the following four activities of a typical software development process: analyse, design, code, and test. These activities are semi-automated with the help of DomainAppTool. Conceptually, the development iterations form a continuous development cycle. Figure 21 illustrates this cycle by an outer curve, which is composted of four curved arrows that connect the four development activities.

There are two dimensions associated to an iteration: *scope* and *refinement*. As far as **scope** is concerned, each iteration is defined within the boundary of a sub-model, so that the output software prototype executes this sub-model. The sub-model of one iteration can be a sub-model that we identified in the previous phase or a smaller sub-model of this sub-model. The exact size of the sub-model depends on the development resources (most importantly the human resource) that are available for the project.

As far as **refinement** is concerned, the sub-model of an interation is incrementally enriched with more detailed features, which include new domain classes, attributes (including the associative ones) and operations. Each domain class can initially be designed with just the default `id` attribute and then gradually updated, through the iterations, with other attributes and operations as the requirements are analysed in more detail.

A typical **iteration design procedure** would proceed as follows:

1. Use the scope dimension to define an initial set of iterations based on the sub-models
2. If necessary, consider the refinement scope. For each iteration in the initial set, break it down into two or more smaller iterations, which refine the domain classes in the iteration's sub-model

Example: CourseMan

This example illustrates how the development iterations of the CourseMan software are defined. For simplicity, we only consider the scope dimension and define the iterations based on their sub-models. The refinement dimension can easily be factored in to further break down the iterations if necessary.

Table 3: Development iterations of the CourseMan software

Iterations	Sub-model(s)	Use case(s)
1	`Student`	Manage students
2	`CourseModule`	Manage course modules
3	`Student`, `CourseModule`	Manage enrolments
4	`SClass`, `Student`	Manage student classes
5	`StudentByNameReport`, `Student`	Manage student-by-name reports

Table 3 lists five development iterations that are defined for CourseMan. These iterations are derived from the five sub-models discussed in the example of Section 7.1. To ease notation in this case, we list the sub-models using only their domain classes.

7.3. Perform the development iterations

We perform each development iteration by carrying out analysis, design, code and test to develop a prototype of a sub-model. The fact that the iterations are performed repeatedly over the software models (functional and domain models) is depicted in Figure 21 by the encapsulation of the development cycle around both the use case model and the domain model. The software prototypes that are produced by the development iterations are depicted on the far right of the figure. There is an arrow, labelled "produce", that points from the development cycle to the software prototypes.

A key feature of the DDSDM's development cycle is that the development iterations can be organised for parallel execution. This is mainly due to the fact that the sub-models of the iterations (though overlap) realise the requirements of different functions. We will illustrate how parallel execution can be achieved in our discussion of the CourseMan example below.

Example: CourseMan

The results of performing the CourseMan's iterations listed in Table 3 were reported in the examples of Chapters 3-6. We briefly illustrate below the outputs of each iteration.

Iteration 1

Table 4: Descriptions of the development activities for iteration 1

Activity	**Description**	**Outcome**
Analyse	Analyse the use case "Manage Students" to discover a new class named `City`. This class represent the address of each `Student`.	A conceptual sub-model with two domain classes `Student` and `City`
Design	Apply the design rules described in Chapter 3 to design `Student` and `City` (complete with attributes and operations).	A detailed design sub-model
Code	• Realise `Student` and `City` in the target language (e.g. Java). • Update the software class `CourseManSoftware` to include these two classes.	• Source code tree containing two packages: `model`, `software` • Code package `model` contains the domain classes of the sub-model • Code package `software` contains the software class (this class also represents the software prototype)
Test	• Execute `CourseManSoftware` with (1) argument `configure` (in the first execution) and (2) argument `run` in the subsequent executions. • Create/update/delete the domain objects of each domain class to test the design of that class	• Test data sheet (one for each class): this includes test cases, expected results, and actual screen captures

Table 4 summarises the four development activities as they are performed in iteration 1. Figures 23-25 show the outcomes of the two key activities: design and code. Figure 23 is the screen capture of the source code tree of the development iteration 1, which is defined in Eclipse. The source code tree consists of two key packages: `model` and `software`. Package `model` consists of all the domain classes of the sub-model of the iteration. Package `software`, on the other hand, consists in the software class that is used to run the software prototype.

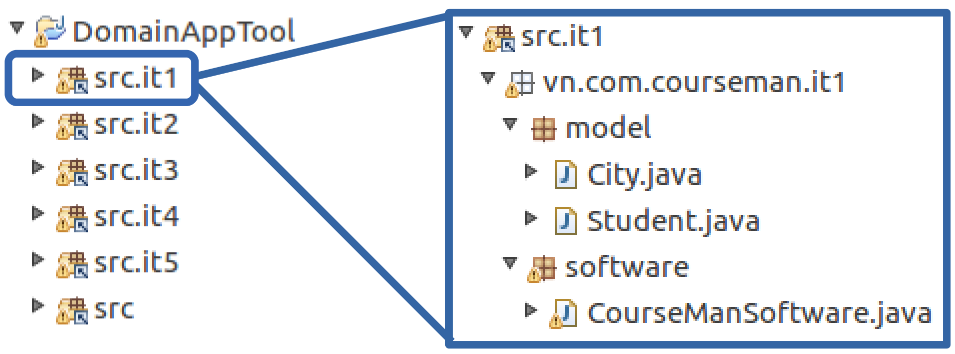


Figure 23: The source code tree of development iteration 1.

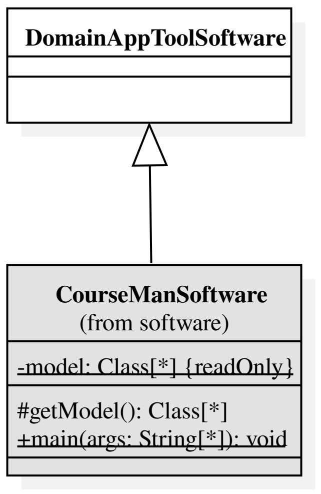
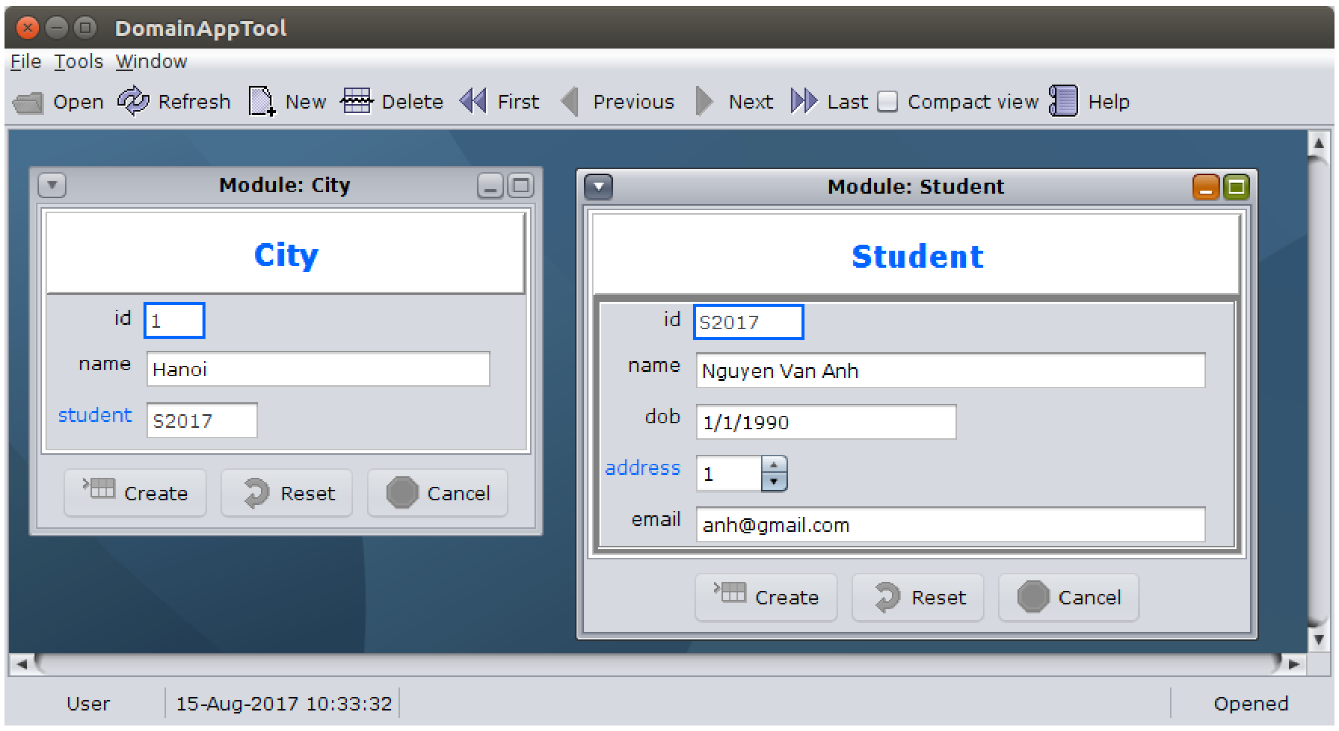


Figure 24: The software class of development iteration 1.



Module: City

id	1
name	Hanoi
student	S2017

Module: Student

id	S2017
name	Nguyen Van Anh
dob	1/1/1990
address	1
email	anh@gmail.com

15-Aug-2017 10:33:32

Figure 25: The software prototype of development iteration 1.

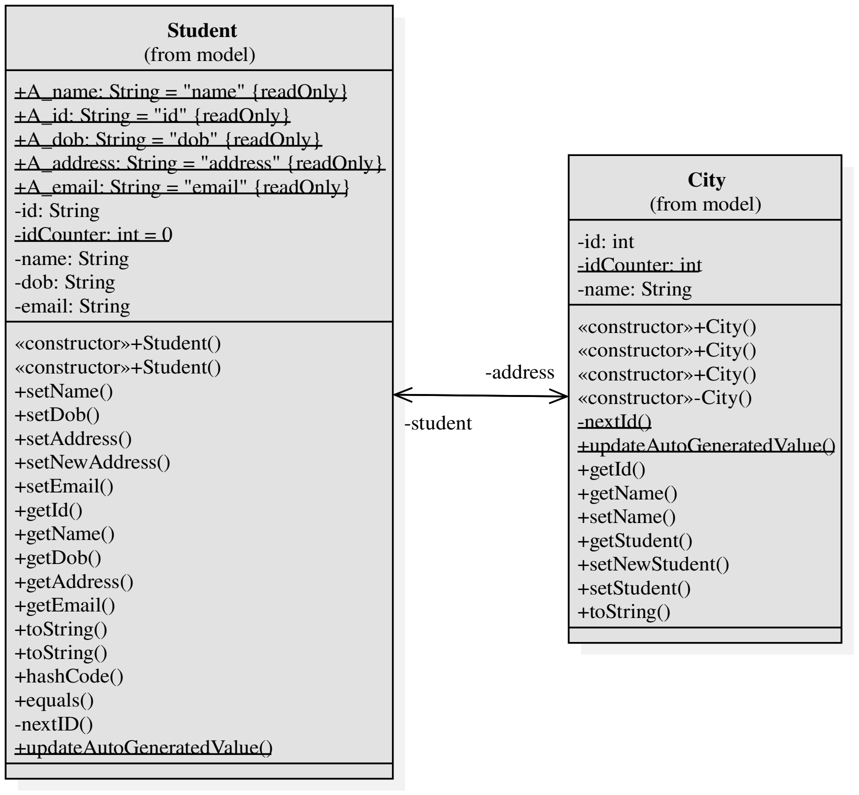


Figure 26: The completed sub-model of development iteration 1.

Figure 26 shows the UML design class diagram of the two domain classes that make up the model of this iteration: `Student` and `City`. Figure 24 shows the design of the software class named `CourseManSoftware` class. Figure 25 shows the GUI of the software prototype when it is executed. The GUI consists of two functional views: one for `City` (displayed on the left hand side) and one for `Student` (displayed on the right hand side).

Notice how the `Student`'s design in Figure 26 as well as the `Student`'s object form in Figure 25 do not include any fields relating to other concepts of the domain model (including `SClass` and `Enrolment`). These fields will be added in the subsequent iterations when we extend the domain model to cover the requirements concerning these concepts.

Iteration 2

Table 5 summarises the four development activities as they are performed in iteration 2.

Table 5: Descriptions of the development activities for iteration 2

Activity	**Description**	**Outcome**
Analyse	Analyse the use case “Manage course modules” to discover two new sub-classes for `CourseModule`.	A conceptual sub-model with three domain classes `CourseModule`, `CompulsoryModule`, `ElectiveModule`.
Design	Apply the design rules described in Chapter 3 to design `CourseModule`, `CompulsoryModule`, `ElectiveModule` (complete with attributes and operations).	A detailed design sub-model
Code	• Realise `CourseModule`, `CompulsoryModule`, `ElectiveModule` in the target language (e.g. Java). • Update the software class `CourseManSoftware` to include these three classes.	• Source code tree containing two packages: `model`, `software` • Code package `model` contains the domain classes of the sub-model • Code package `software` contains the software class (this class also represents the software prototype)
Test	• Execute `CourseManSoftware` with (1) argument `configure` (in the first execution) and (2) argument `run` in the subsequent executions. • Create/update/delete the domain objects of each domain class to test the design of that class	• Test data sheet (one for each class): this includes test cases, expected results, and actual screen captures

Figures 27-29 show the outcomes of the two key activities: design and code. Figure 27 is the screen capture of the source code tree of development iteration 2. This source code tree has a similar structure to that of the previous iteration. Package `software`, in particular, contains a software class for running the software prototype. Since this class has a similar design to that of the previous iteration, its details are omitted.

Figure 28 shows the design of the three domain classes of the sub-model of the iteration:

`CourseModule`, `CompulsoryModule`, and `ElectiveModule`. The latter two classes are added as the result of analysing the requirement of the associated use case. Figure 29 shows the GUI of the software prototype of the iteration. This GUI consists of three functional views, one for each domain class.

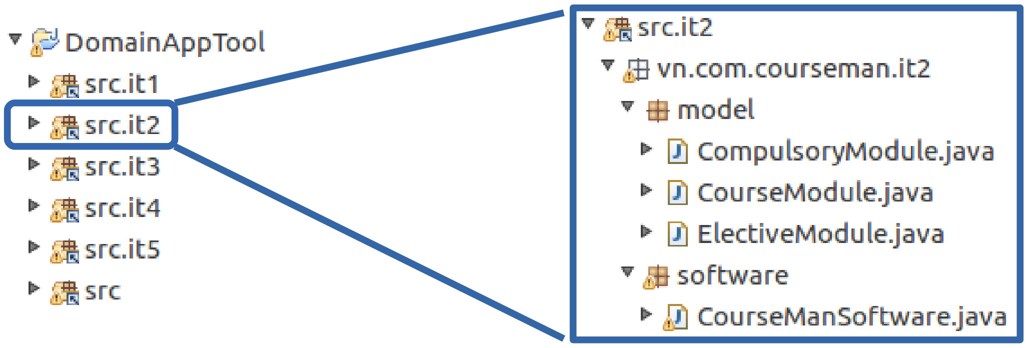


Figure 27: The source code tree of development iteration 2.

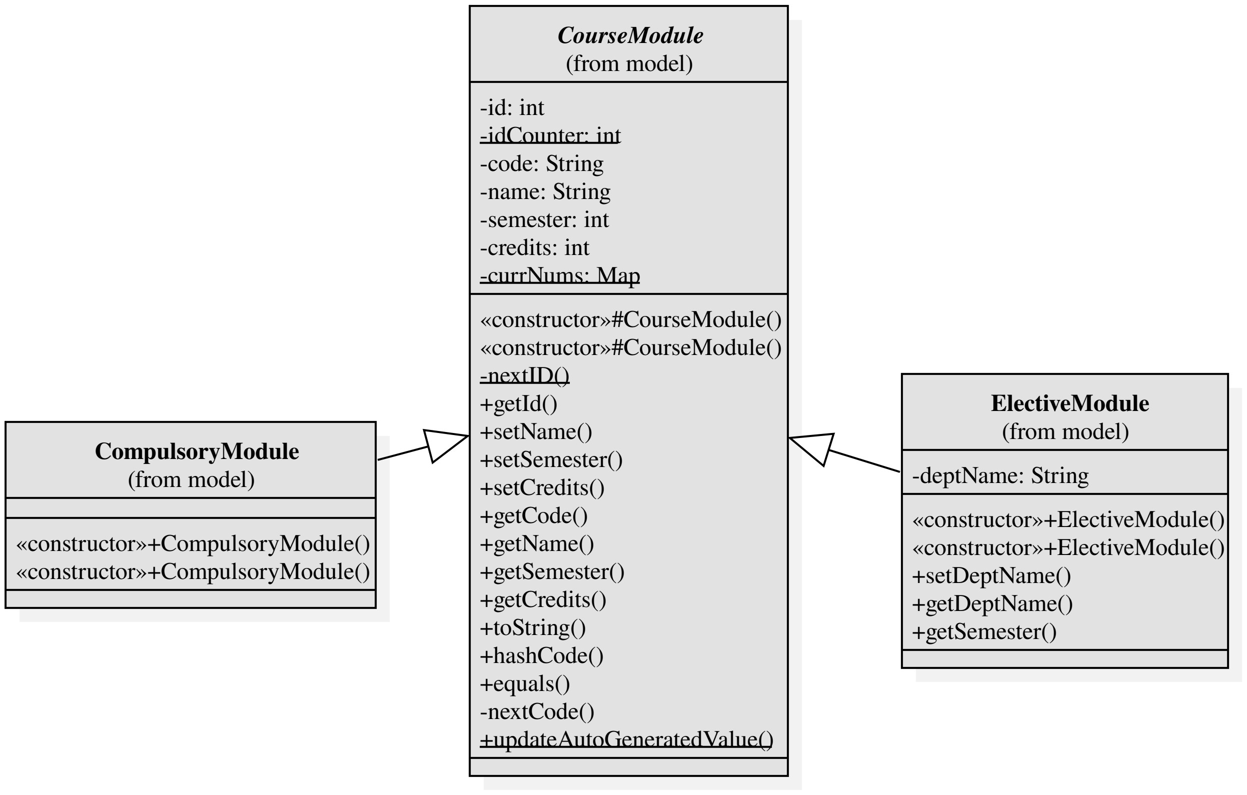
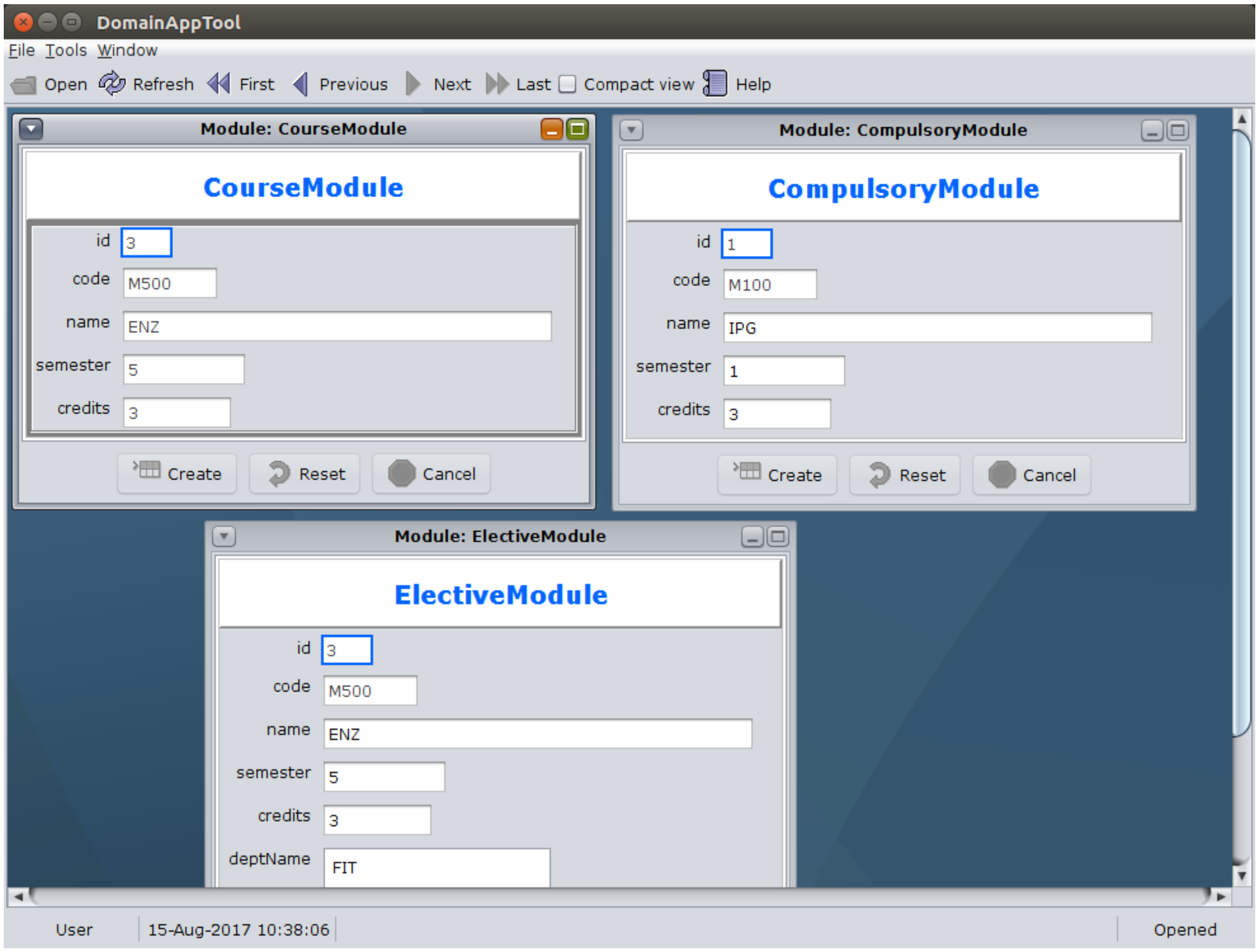


Figure 28: The completed sub-model of development iteration 2.



Module: CourseModule

CourseModule

id	3
code	M500
name	ENZ
semester	5
credits	3

Module: CompulsoryModule

CompulsoryModule

id	1
code	M100
name	IPG
semester	1
credits	3

Module: ElectiveModule

ElectiveModule

id	3
code	M500
name	ENZ
semester	5
credits	3
deptName	FIT

Figure 29: The software prototype of development iteration 2.

Iteration 3

Table 5 summarises the four development activities as they are performed in iteration 3. Figures 30-32 show the outcomes of the two key activities: design and code.

Figure 30 shows the source code tree of development iteration 3. Package `model` contains the five domain classes from the previous two iterations and one new domain class named `Enrolment`. This class is added to realise the requirement concerning student enrolments into course modules.

Figure 31 shows the sub-model of the iteration. To ease reading, we only highlight and show the design details of the class `Enrolment`, because this class is the primary design subject of this iteration.

A question could be raised as to why we need all five classes from the first iteration, if all we need to form `Enrolments` are `Student` and `CourseModule`? There are two main reasons here.

The first reason concerns the dependencies between Student and City. Although City is not directly involved in the definition of Enrolment, its association to Student requires that it is included in the sub-model. The second reason concerns the inclusion of CompulsoryModule and ElectiveModule. These classes are included simply because CourseModule is abstract and thus cannot be used to create objects. We need CompulsoryModule and ElectiveModule in order to create course module objects that can be used for testing. If CourseModule was not abstract, we would not need these two classes.

Note that classes Student and CourseModule do not have to be completely finished in order for Enrolment to be realised in this iteration's sub-model. The same argument also applies to the other three classes. All that is required is that these classes are designed with a suitable id attribute. This helps plan the iterations so that they can be performed in parallel. For example, we can plan the first three iterations of CourseMan so that they can be performed in parallel as follow. First, we design a simple version of the three classes Student, CourseModule, and City. Then we distribute these classes to the development groups that work in parallel on the first three iterations. As these groups carry out the iterations, the three classes are independently updated with new features without causing any interference among the groups.

Figure 32 shows the GUI of the software prototype of this iteration. For brevity, we only show the two key functional views of these iteration: Enrolment's and Student's view. The former shows the Enrolment objects, while the latter shows the association between Student and Enrolment.

Table 6: Descriptions of the development activities for iteration 3

Activity	Description	Outcome
Analyse	Analyse the use case "Manage enrolments" to discover a new class named Enrolment.	A conceptual sub-model with three main domain classes Student, CourseModule, Enrolment.

Activity	Description	Outcome
Design	Apply the design rules described in Chapter 3 to update `Student` with an association to `Enrolment` and to create `Enrolment` (complete with attributes and operations).	A detailed design sub-model
Code	• Import domain classes from the previous iterations to resolve any necessary dependencies. • Realise `Student`, `CourseModule`, `Enrolment` in the target language (e.g. Java). • Update the software class `CourseManSoftware` to include the above classes.	• Source code tree containing two packages: `model`, `software` • Code package `model` contains the domain classes of the sub-model • Code package `software` contains the software class (this class also represents the software prototype)
Test	• Execute `CourseManSoftware` with (1) argument `configure` (in the first execution) and (2) argument `run` in the subsequent executions. • Create/update/delete the domain objects of each domain class to test the design of that class	• Test data sheet (one for each class): this includes test cases, expected results, and actual screen captures

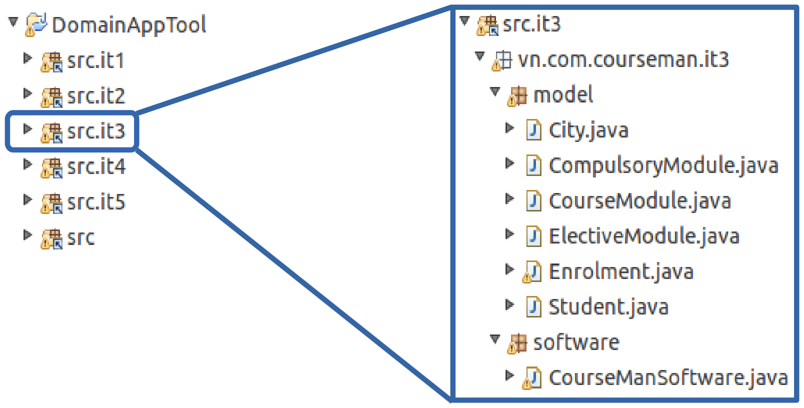


Figure 30: The source code tree of development iteration 3.

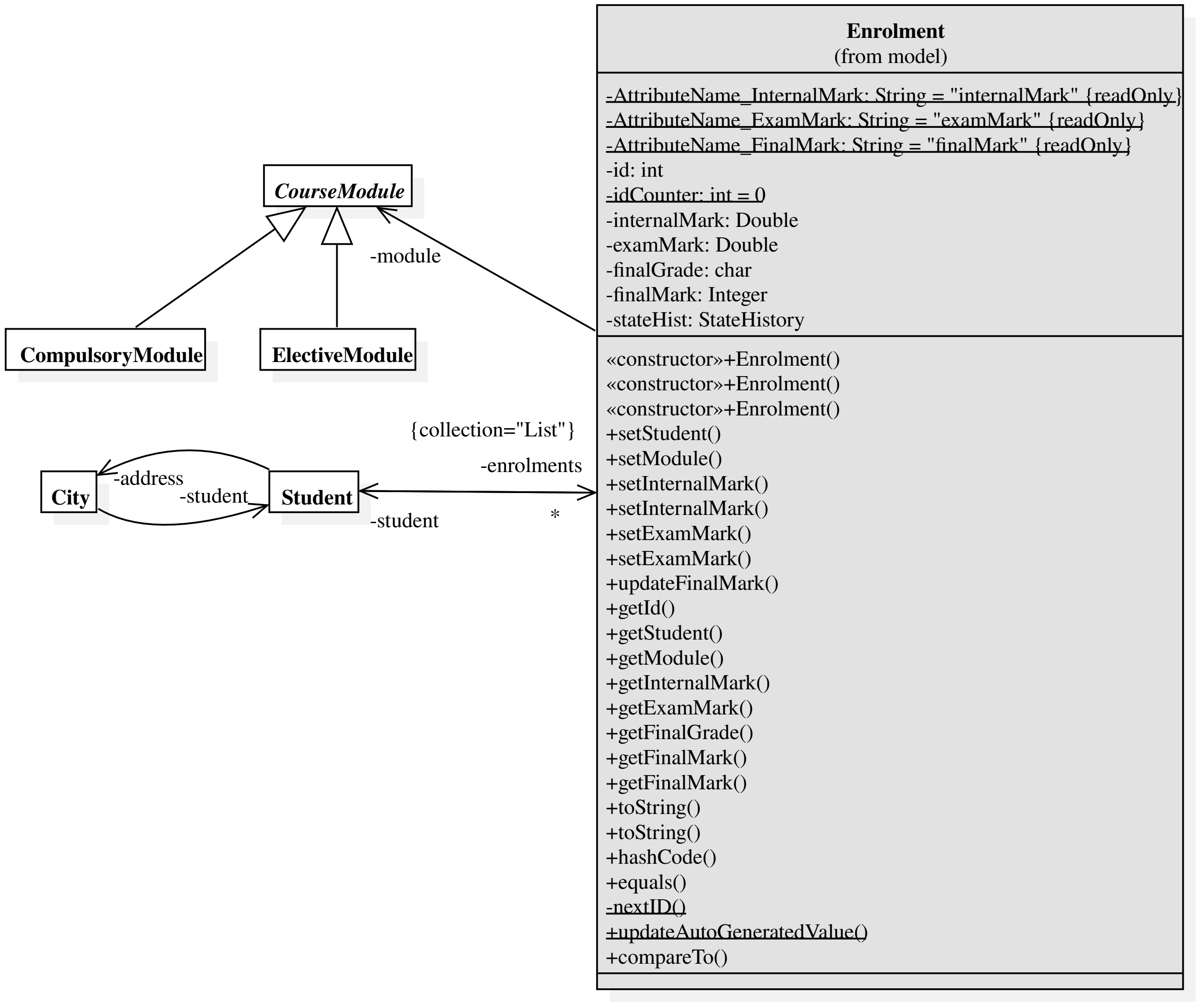
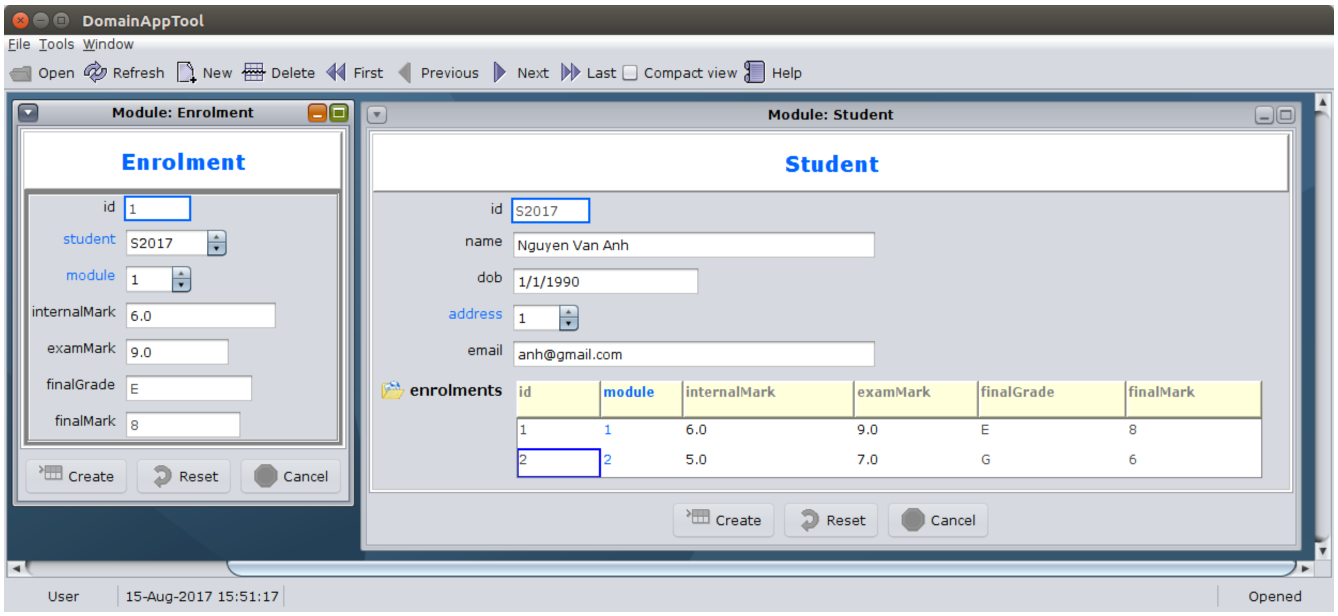


Figure 31: The completed sub-model of development iteration 3.



Module: Enrolment

Enrolment

id	1
student	S2017
module	1
internalMark	6.0
examMark	9.0
finalGrade	E
finalMark	8

Module: Student

Student

id	S2017
name	Nguyen Van Anh
dob	1/1/1990
address	1
email	anh@gmail.com

enrolments

id	module	internalMark	examMark	finalGrade	finalMark
1	1	6.0	9.0	E	8
2	2	5.0	7.0	G	6

User 15-Aug-2017 15:51:17 Opened

Figure 32: The software prototype of development iteration 3.

Iteration 4

Table 7 summarises the four development activities as they are performed in iteration 4. Figures 33-35 show the outcomes of the two key activities: design and code.

Figure 33 shows the source code tree of the development iteration 4. Package `model` contains six domain classes of this iteration's sub-model. Class `SClass` is a new domain class that realises the student class concept. The other five classes realise the concepts that are developed in the previous three iterations.

Figure 34 shows the design of the sub-model. For brevity, we only show details of the class `SClass`. The other five classes are drawn using a simple labelled box. Among these five classes, only class `Student` is required to directly realise the association with `SClass`. The other four classes are included in the sub-model only to satisfy the dependencies that are either directly and indirectly associated to `Student`.

Based on a similar argument to the one that was made in iteration 3, we further argue that these five classes need not be completely finished before they can be included in the sub-model of this iteration. This allows the development team to plan this iteration so that it is, at least to a certain extent, executed in parallel to the other iterations.

Figure 35 shows the GUI of the software prototype of this iteration. For brevity, we only include in the figure two functional views: one for `SClass` and the other is for `Student`. As can be seen from this figure, the `SClass`'s object form includes a sub-form for the `Students` that belong to a given `SClass`. This sub-form depicts the association between `SClass` and `Student`. It has the exact same layout as the `Student`'s object form.

Table 7: Descriptions of the development activities for iteration 4

Activity	**Description**	**Outcome**
Analyse	Analyse the use case "Manage student classes" to discover a new class named `SClass`.	A conceptual sub-model with two main domain classes `Student`, `SClass`.
Design	Apply the design rules described in Chapter 3 to update `Student` with an association to `SClass` and to create `SClass` (complete with attributes and operations).	A detailed design sub-model
Code	• Import domain classes from the previous iterations to resolve any necessary dependencies. Realise `Student`, `SClass` in the target language (e.g. Java). Update the software class `CourseManSoftware` to include the above classes.	• Source code tree containing two packages: `model`, `software` Code package `model` contains the domain classes of the sub-model Code package `software` contains the software class (this class also represents the software prototype)
Test	• Execute `CourseManSoftware` with (1) argument `configure` (in the first execution) and (2) argument `run` in the subsequent executions. Create/update/delete the domain objects of each domain class to test the design of that class	• Test data sheet (one for each class): this includes test cases, expected results, and actual screen captures

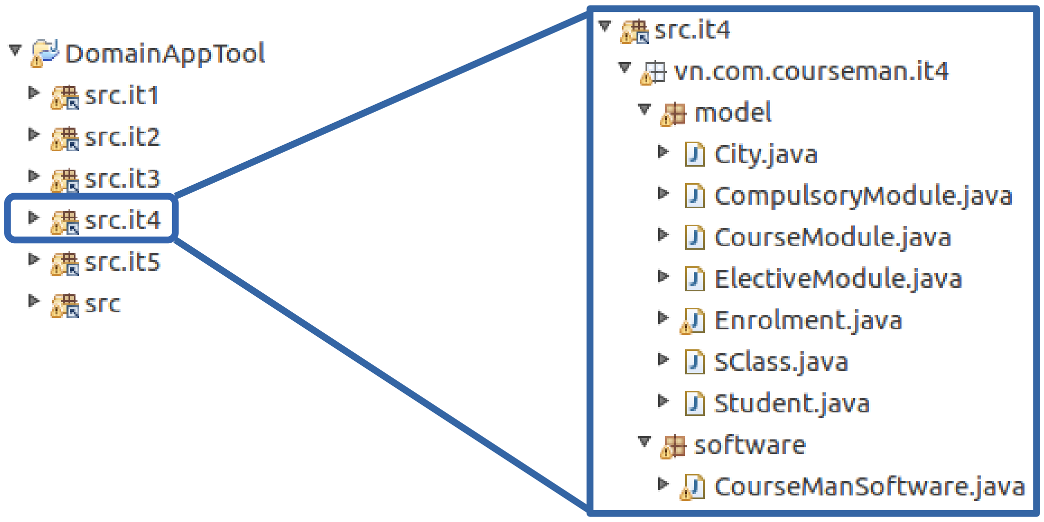


Figure 33: The source code tree of development iteration 4.

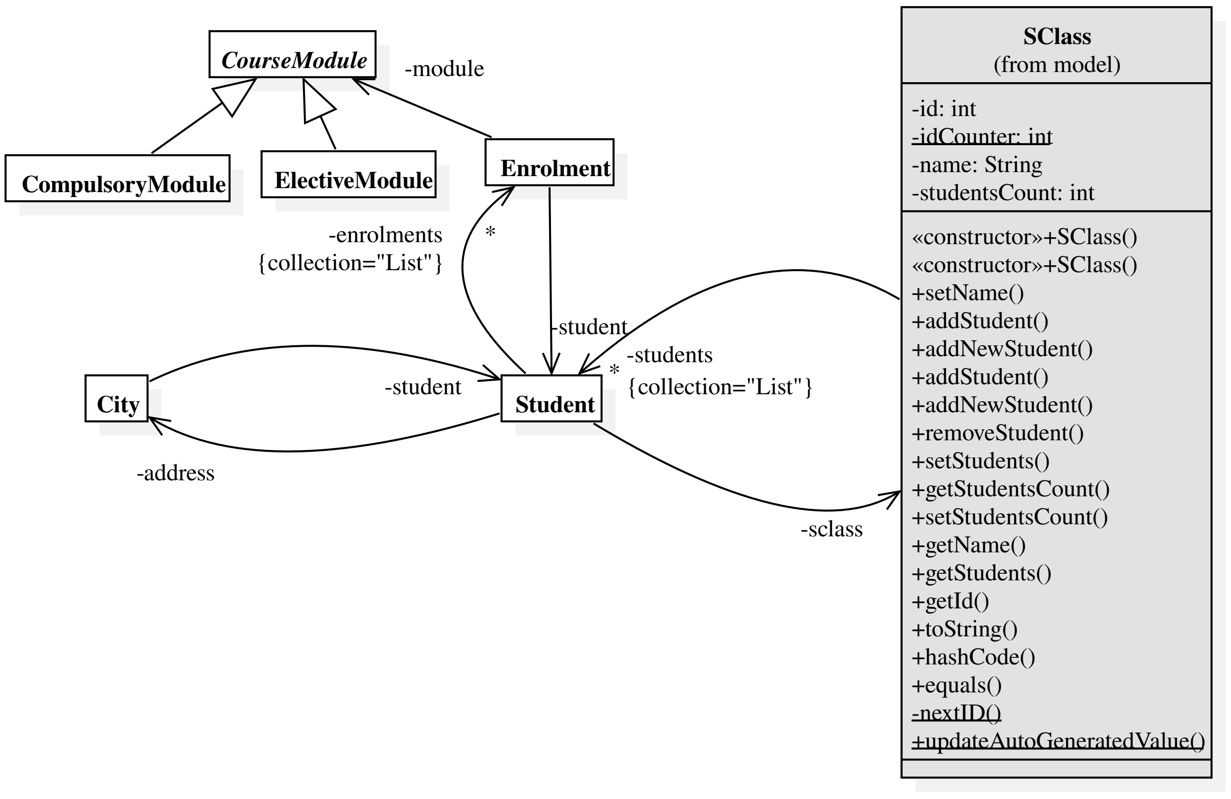
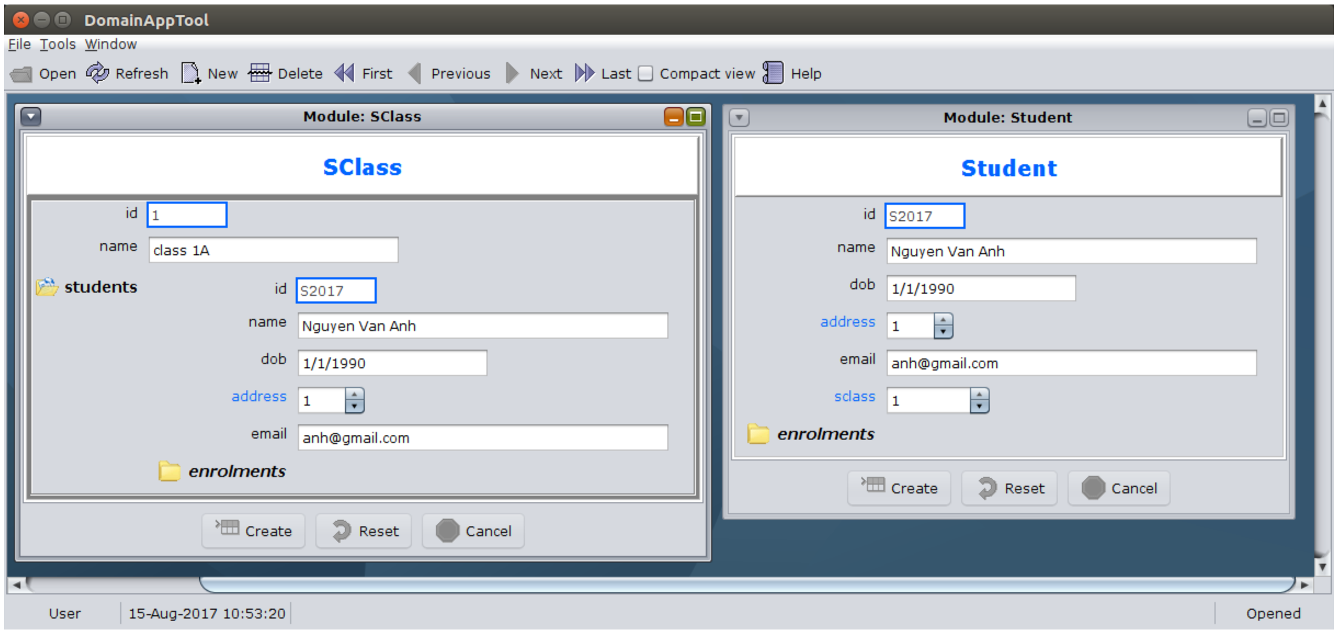


Figure 34: The completed sub-model of development iteration 4.



SClass

id: 1
name: class 1A
students
id: S2017
name: Nguyen Van Anh
dob: 1/1/1990
address: 1
email: anh@gmail.com
enrolments

Student

id: S2017
name: Nguyen Van Anh
dob: 1/1/1990
address: 1
email: anh@gmail.com
sclass: 1
enrolments

15-Aug-2017 10:53:20

Figure 35: The software prototype of development iteration 4.

Iteration 5

Table 8 summarises the four development activities as they are performed in iteration 5. Figures 36-38 show the outcomes of the two key activities: design and code.

Figure 36 shows the source code tree of the development iteration 5. In addition to containing all of the domain classes from the other iterations, package `model` contains a sub-package named `reports`. This package includes the domain class named `StudentByNameReport`, which represents the report that we described in the examples of Chapter 6.

Figure 37 shows the UML design of the sub-model of this iteration. In particular, we highlight the design of the class `StudentByNameReport`. Figure 38 shows the GUI of the software prototype of this iteration. For brevity, we shows only the functional view of `StudentByNameReport`.

Table 8: Descriptions of the development activities for iteration 5

Activity	**Description**	**Outcome**
Analyse	Analyse the use case “Manage student-by-name reports” to discover a new class named `StudentByNameReport`.	A conceptual sub-model with two main domain classes `StudentByNameReport`, `Student`.

Activity	Description	Outcome
Design	• Apply the design rules described in Chapter 6 to create `StudentByNameReport` (complete with attributes and operations). • Apply the design rules described in Chapter 3 to update `Student` with an association to `StudentByNameReport`.	A detailed design sub-model
Code	• Import domain classes from the previous iterations to resolve any necessary dependencies. • Realise `Student` in package `model` and `StudentByNameReport` in package `reports`. • Update the software class `CourseManSoftware` to include the above classes.	• Source code tree containing three packages: `reports`, `model`, `software` • Package `reports` contains the report domain class • Package `model` contains the other domain classes of the sub-model • Package `software` contains the software class (this class also represents the software prototype)
Test	• Execute `CourseManSoftware` with (1) argument `configure` (in the first execution) and (2) argument `run` in the subsequent executions. • Create/update/delete the domain objects of each domain class to test the design of that class	• Test data sheet (one for each class): this includes test cases, expected results, and actual screen captures

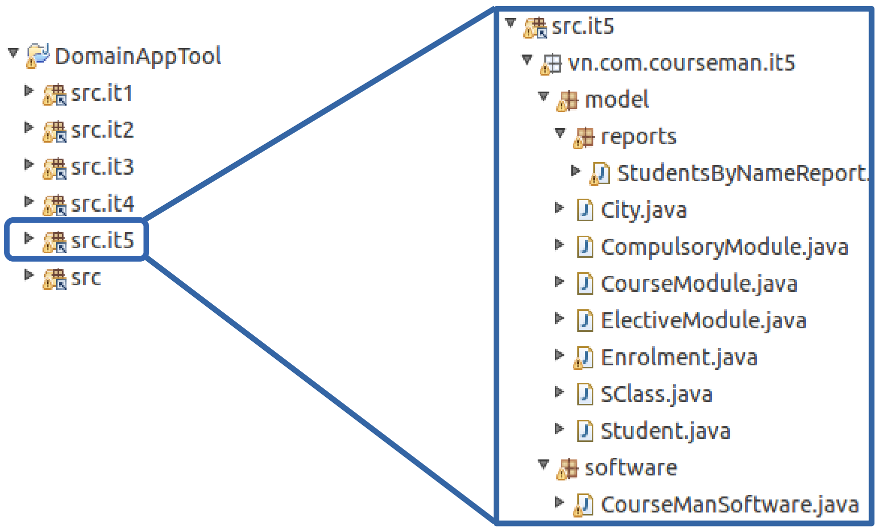


Figure 36: The source code tree of development iteration 5.

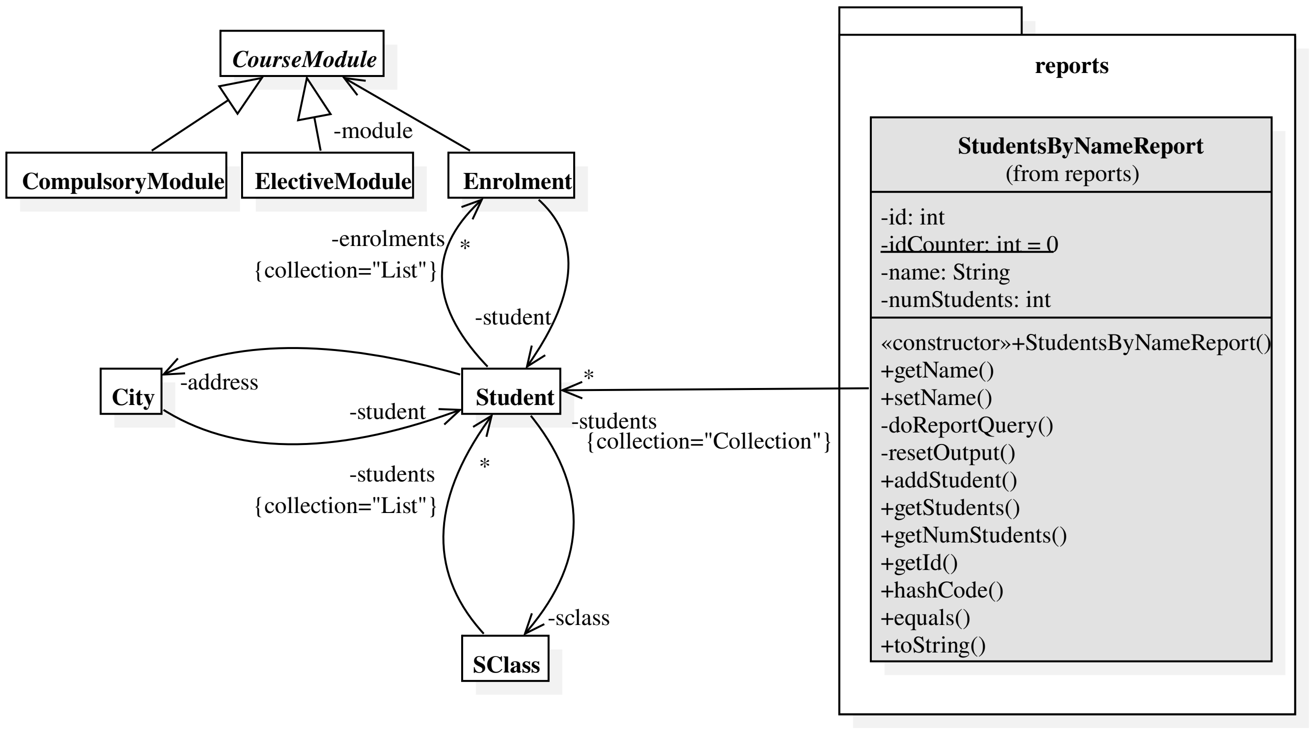
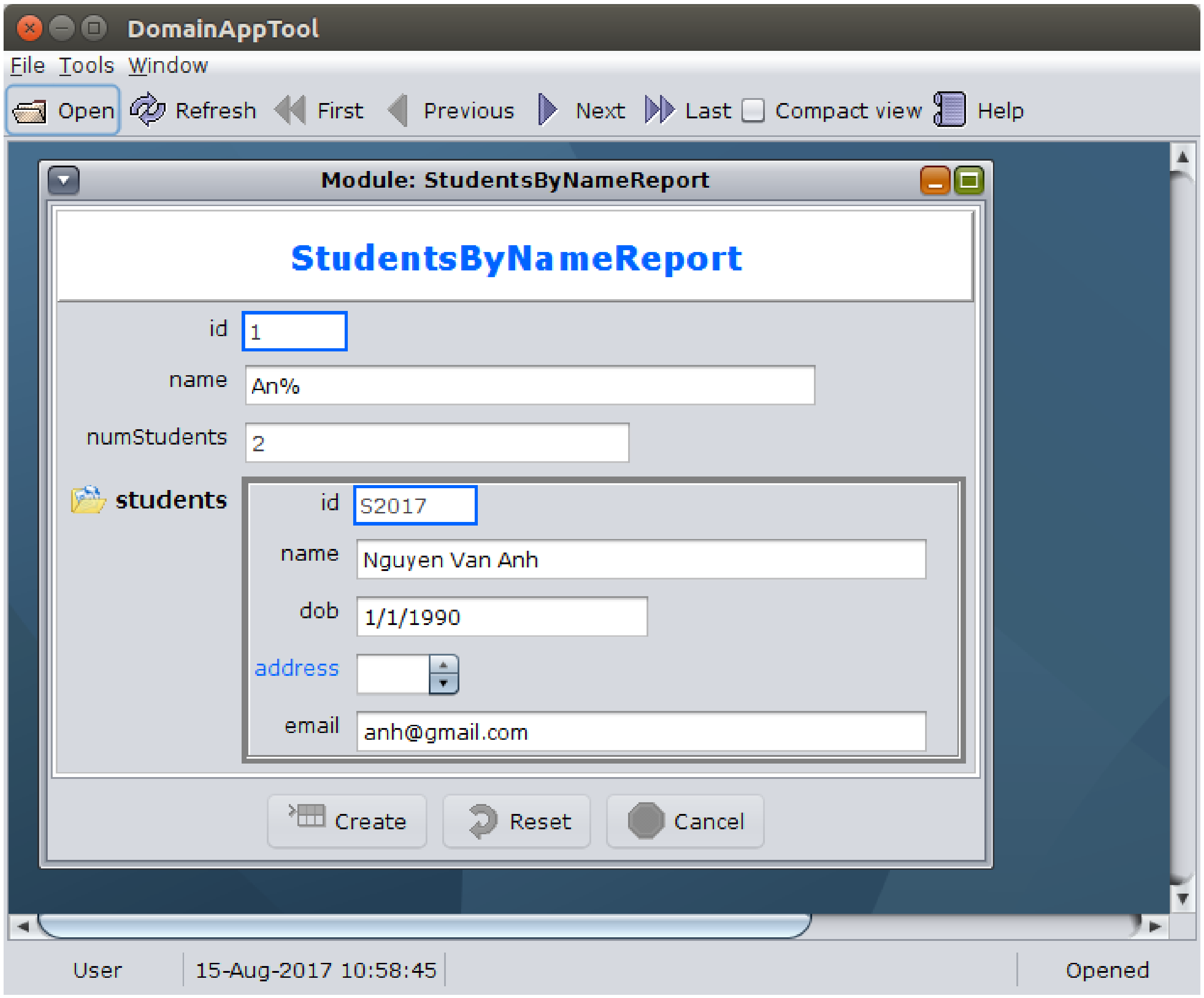


Figure 37: The completed sub-model of development iteration 5.



Module: StudentsByNameReport

StudentsByNameReport

id: 1
name: An%
numStudents: 2
students
id: S2017
name: Nguyen Van Anh
dob: 1/1/1990
address
email: anh@gmail.com

User | 15-Aug-2017 10:58:45 | Opened

Figure 38: The software prototype of development iteration 5.

7.4. Integrate the software prototypes

Once we have completed the development iterations which produce a set of prototypes for the sub-models, the final phase is to integrate these prototypes. The objectives of this integration are two-folds: (1) to produce the final domain model and (2) to produce the complete software prototype.

Software integration is achieved with ease in DDSDM thanks to the DomainAppTool's ability to directly support the domain-driven design of the sub-models. As the sub-models are enriched through the iterations, they are merged together using the shared domain classes and/or through the identification of new associations that associate the domain classes from the sub-models.

Because all software functions have already been accounted for by the iterations, it is unnecessary to carry out any analysis, design and code in this phase. That leaves us with testing being the only activity that needs to be performed. The objective of testing is to run DomainAppTool on the complete domain model to ensure that the overall software prototype functions correctly (*i.e.* all of its parts are working together correctly).

Figure 21 explains integration by an arrow, labelled "integrate", that connects the software prototypes to a final prototype. Successfully testing this prototype also completes the domain model of the software.

Example: CourseMan

As it was evident in our discussions of the five development iterations in Section 7.3, although the sub-models of the iterations involve different subsets of the domain classes, together these classes form a common source code tree (of the CourseMan's domain model). Technically, when we work on the same domain class in two different iterations, we are updating this class with new features that realise the requirements of the iterations. Assuming that the two feature sets do not conflict (or may contain conflicts but these conflicts can satisfactorily be resolved), we progress towards a complete domain model of CourseMan.

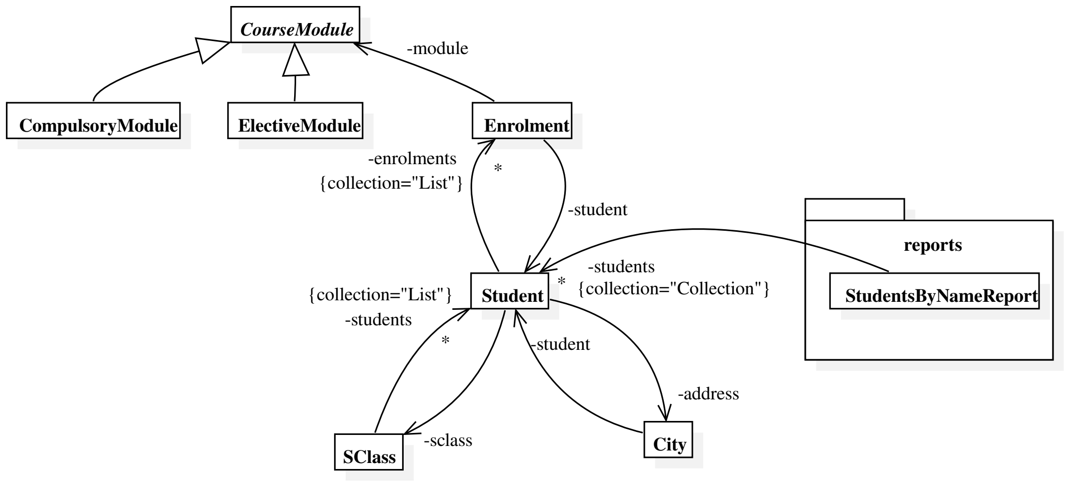


Figure 39: The complete (final) domain model of CourseMan.

Figure 39 shows the complete domain model of CourseMan. For brevity, we omit the attributes and operations of the domain classes. Please refer to the sub-models of the development iterations for these. It is straight-forward to see that this domain model contains the domain classes from both Figure 10 and Figure 16.

Exercises

Unlike other chapters, this chapter does not provide any specific exercises because these exercises are defined by the students! This is the point where the students start working on their projects. The primary goal is apply the DDSDM method described in this chapter and the techniques defined in the previous chapters to develop the domain model and software to solve a real-world problem. A general requirement is that students must be able to break their software development process down into *at least three iterations* and write a technical report detailing the process.

Students are expected to investigate the "hot" problem domain or domains that are provided by the course instructor in order to come up with a problem that each group is interested in solving. Please refer to the section titled "Course policies and expectations" in the Preface chapter of this book for some guidance and an example on how to achieve this.

Bibliography

[1] T. Rentsch, “Object Oriented Programming,” *SIGPLAN Not*, vol. 17, no. 9, pp. 51–57, Sep. 1982, doi: 10.1145/947955.947961.
[2] G. Booch, “Object-Oriented Development,” *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 211–221, Feb. 1986, doi: 10.1109/TSE.1986.6312937.
[3] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Santa Barbara (California): ISE Inc., 1997.
[4] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Pearson Education, 2000.
[5] 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.
[6] D. H. H. Ingalls, “Design Principles Behind Smalltalk,” *BYTE Mag.*, p. 7, Aug. 1981.
[7] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.
[8] Apache-S.F, “Apache Isis,” 2016. [Online]. Available: http://isis.apache.org/. [Accessed: 05-May-2016].
[9] Dan Haywood, “Apache Isis - Developing Domain-Driven Java Apps,” *Methods Tools Pract. Knowl. Source Softw. Dev. Prof.*, vol. 21, no. 2, pp. 40–59, 2013.
[10] R. Pawson and R. Matthews, “Naked Objects,” in *OOPSLA ’02*, New York, NY, USA, 2002, pp. 36–37, doi: 10.1145/985072.985091.
[11] “OpenXava,” 2016. [Online]. Available: http://openxava.org/. [Accessed: 05-May-2016].
[12] J. Paniza, *Learn OpenXava by Example*. CreateSpace Independent Publishing Platform, 2011.
[13] D. M. Le, “jDomainApp version 5.0: A Java Domain-Driven Software Development Framework,” Hanoi University, 2017.
[14] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 4th ed., vol. 2. Addison-wesley Reading, 2005.
[15] J. Gosling, B. Joy, G. L. S. Jr, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.
[16] 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, doi: https://doi.org/10.1109/KSE.2015.26.
[17] D. M. Le, “A Domain-Oriented, Java Specification Language,” in *Proc. 7th Int. Conf. Knowledge and Systems Engineering (KSE)*, 2015, pp. 25–30, doi: https://doi.org/10.1109/KSE.2015.27.
[18] 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, doi: https://doi.org/10.1109/KSE.2016.7758031.

[19] D. M. Le, D.-H. Dang, and V.-H. Nguyen, “Domain-Driven Design Patterns: A Metadata-Based Approach,” in *Proc. 12th Int. Conf. on Computing and Communication Technologies (RIVF)*, 2016, pp. 247–252, doi: https://doi.org/10.1109/RIVF.2016.7800302.

[20] D. M. Le, D.-H. Dang, and V.-H. Nguyen, “On Domain Driven Design Using Annotation-Based Domain Specific Language,” *J. Comput. Lang. Syst. Struct.*, vol. 54, pp. 199–235, 2018, doi: https://doi.org/10.1016/j.cl.2018.05.001.

[21] OMG, “Unified Modeling Language version 2.5,” OMG, formal/2015-03-01, 2015.

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

[23] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[24] Stanford University, “Creating A Syllabus,” *Stanford Teaching Commons*, 2018. [Online]. Available: https://teachingcommons.stanford.edu/resources/course-preparation-resources/creating-syllabus. [Accessed: 08-Oct-2018].

[25] D. M. Le, *DomainAppTool*. 2018.

[26] J. Brownlee, “Supervised and Unsupervised Machine Learning Algorithms,” *Machine Learning Mastery*, 15-Mar-2016. [Online]. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. [Accessed: 09-Oct-2018].

[27] Oracle, “Java DB Reference Manual,” 2014. [Online]. Available: http://docs.oracle.com/javadb/10.10.1.2/ref/index.html. [Accessed: 16-Jun-2017].

[28] Oracle, “Getting Started with Java DB,” 2014. [Online]. Available: http://docs.oracle.com/javadb/10.10.1.2/getstart/index.html. [Accessed: 16-Jun-2017].

[29] V. Cepa and S. Kloppenburg, “Representing Explicit Attributes in UML,” in *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

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

[31] J. A. Hoffer, R. Venkataraman, and H. Topi, *Modern Database Management*, 10th ed. Upper Saddle River, N.J: Prentice Hall, 2010.

Appendix 1. Java code of the CourseMan software

See the attached domain model source code file of CourseMan: `courseman-source.zip`.

Appendix 2. How to create a DomainAppTool project in Eclipse

A software project that uses DomainAppTool is a Java project, which imports the DomainAppTool binary and libraries. We list below the essential steps for creating such a project in Eclipse:

1. Create a Java project.

   Ensure that the "Execution environment JRE" setting points to a Java **JDK** installation. Simple JRE installations will *not* work.

2. Add to the project's library the DomainAppTool binary and other `.jar` files provided as part of the distribution

3. Create two top-level packages in your project's source code: `model` and `software`

   1. `model`: contains the domain classes that make up the domain model (see Chapter 3)

   2. `software`: contains an implementation of the software class of your project (see Section 5.4)

Appendix 3. Syllabus of the SS2 course module

We list below the most current syllabus of the SS2 course module that we currently used in the semester of Spring 2018. We used this syllabus in this book to formulate a model syllabus for course modules that are based on the book.

HANOI UNIVERSITY
Faculty of Information Technology

FIT447

Special Subject 2

Spring 2018

Module Description

Contact details

Lecturer**:**	Le, Minh Duc
	Nguyen, Van Cong
Office:	Building C, Room 210, Hanoi University
Tel:	8544338 (ext 3236)
Email:	le_m_duc@gmail.com
	congnv8192@gmail.com
	(use the module's news forum for all teaching-related matters)

Unit overview

1. Unit details

Faculty	Information Technology
Module code	FIT447
Module name	**Special Subject 2**
Level	Undergraduate
Credits	Three (3)
Prerequisites	FIT346, FIT330
Suggested study	at least 6 hours of self-study each week
Year	2018

2. Nature of the module

This module provides students with a hands-on practical experience in building a real-world software using a modern software framework. Students will first study a software development tool named DomainAppTool, which is the developer interface of a Java software framework. Next, students will form into small groups to investigate and define a software development problem suitable for the time available in the module. The students will then use DomainAppTool, together with the software engineering and IT-related knowledge and skills that they learnt in previous modules, to develop and evaluate a software solution for the problem. The groups will report their progress and receive feedbacks and guidance from the teaching staff in the weekly meetings.

3. Learning objectives

Upon completion of this module, students are expected to:

3.1 Study to understand a modern software framework and tool for developing software.
3.2 Investigate and define the requirements for a software-based problem.
3.3 Apply a software framework and tool to design, implement, and test a software solution for a defined problem.
3.4 Propose an improvement feature for the software framework.

Learning Resources

[1] D. M. Le, "DomainAppTool v5.1: A Domain-Driven Software Development Tool," Hanoi University, Hanoi, 2017.

[2] B. Liskov and J. Guttag, *Program development in Java: abstraction, specification, and object-*

oriented design. Pearson Education, 2000.
[3] I. Sommerville, *Software Engineering*, 9th ed. Pearson, 2011.
[4] Oracle, "Java Platform Standard Edition 8 API Specification." [Online]. Available: http://docs.oracle.com/javase/8/docs/api/index.html. [Accessed: 18-Oct-2016].
[5] OMG, "Unified Modeling Language version 2.5," 2015.

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

Primary resource

The primary teaching resource is a course book about DomainAppTool ([1]) that is written by the lecturer Duc Minh Le of this course module.

Secondary resources

Selected text books ([2]-[6]) that were used in previous course modules.

Prescribed text books

There are no prescribed text books for this module. Other relevant materials will be posted on the course website.

Assessment

1. Assessment table

Assessment	Weight	Brief Description	Learning objectives
Attendance	10%	Weekly attendance in lectures and tutorials	
Practicals	30%	Seminar (max mark: 8/10) 5% Program & Technical report 25%	3.1-3
Final exam	60%	Presentation and demonstration of the program; interview	3.1-3

2. Determination of the final grade

The following assessment rules complement or specialise those in the faculty's standard assessment procedure:

- there is **no retake exam**
- the result of the final exam will be used, together with other assessment items, to determine the internal mark and final grade

- failure in the final exam will automatically lead to failure of the module
- if you failed the module then your internal mark is < 50% and you have no final grade

3. Seminars

Starting from week 6, there is one student seminar held every week for the students to report their progress and to discuss their software development issues.

Proposed weekly schedule

Weeks	Descriptions	Tutorials
1	**Introduction** to the module, topic list **DomainAppTool: Introduction** *Ref*: Chapters 1-3	(No tutorial)
2	**DomainAppTool: Domain class development (1)** *Ref*: Chapters 4.1-4, 5-7	Domain class development exercises (1)
3	**DomainAppTool: Domain class development (2)** *Ref*: Chapters 4.5-4.6.2	Domain class development exercises (2)
4	**DomainAppTool: Domain class development (3)** *Ref*: Chapters 4.6.3-4.7	Domain class development exercises (3)
5	**DomainAppTool (4): Report class development** *Ref*: Chapter 8	Report class development exercises
6	**Problem research** *Ref:* (Internet-based search & software engineering and IT-related knowledge from previous modules)	Problem research report
7	**Software development method** *Ref*: Chapter 9	Development iterations report
8	**Development iteration (1):** analyse, design, code, and test a **micro prototype**	Software prototype (1)
9	**Development iteration (2):** analyse, design, code, and test a **small prototype**	Software prototype (2)
10	**Development iteration (3):** analyse, design, code, and test a prototype for **sub-model 1**	Software prototype (3)
11	**Development iteration (4):** analyse, design, code, and test a prototype for **sub-model 2**	Software prototype (4)
12	**Development iteration (5):** analyse, design, code, and test a prototype for **sub-model 3**	Software prototype (5)
13	**Development iteration (6):** analyse, design, code, and test a prototype for **sub-model 4**	Software prototype (6)
14	**Development iteration (7): Software integration (1)**	Software prototype (7)

Weeks	Descriptions	Tutorials
15	**Development iteration (8): Software integration (2)**	Final software prototype

Software project guidelines

Vietnam is well known for the diversity and quality of its agricultural products and services. In recent years, the Vietnamese government has strongly been promoting the use of and investing in modern agricultural technology. The investment appears to start baring fruits. According to a recent study[12], in 2017 and for the first time, agricultural export has taken over oil export to become the top contributor to the Vietnam's economy. Vietnam's agricultural products are currently exported to 180 countries in the world. More recently, agriculture has been considered[13] one of the three core areas in Vietnam's strategy for the Industrial Revolution 4.0.

Your task in this semester is to investigate and choose one or a group of related architectural products/services for your software development project. The aim is to produce a software that would help improve the effectiveness of various activities that are performed in the life cycle of the chosen products/services.

In addition to using the software framework listed in the teaching plan to develop your software, you need to propose at least one improvement feature for this framework.

12 see http://vietnamnews.vn/politics-laws/416671/agriculture-leads-all-exports.html

13 see http://nhandan.com.vn/nhandan.com.vn/kinhte/item/35358302-co-hoi-vang-tu-cuoc-cach-mang-cong-nghiep-4-0-ky-1.html