On Domain Driven Design Using Annotation-Based Domain Specific Language

Duc Minh Le[a,b], Duc-Hanh Dang[a,*], Viet-Ha Nguyen[a]

[a] *Department of Software Engineering, VNU University of Engineering and Technology, Vietnam*
[b] *Department of Software Engineering, Hanoi University, Vietnam*

Abstract

The aim of object-oriented domain-driven design (DDD) is to iteratively develop software around a realistic domain model. Recent work in DDD use an annotation-based extension of object-oriented programming language to build the domain model. This model becomes the basis for a ubiquitous language and is used as input to generate software. However, the annotation-based extensions of these work do not adequately address the primitive and essential structural and behavioural modelling requirements of practical software. Further, they do not precisely characterise the software that is generated from the domain model. In this paper, we propose a DSL-based DDD method to address these limitations. We make four contributions: (1) An annotation-based domain-specific language (DSL) named DCSL, whose annotation extension expresses a set of essential structural constraints and the essential behaviour of a domain class. (2) A structural mapping between the state and behaviour spaces of a domain class. This mapping enables a technique for generating the behavioural specification. (3) A technique that uses DCSL to support behavioural modelling with UML activity diagram. (4) A 4-property characterisation of the software generated from the domain model. We demonstrate our method with a Java software tool and evaluate DCSL in the context of DDD.

Keywords: Domain-driven design (DDD); UML-based domain modelling; Domain-specific language (DSL); Object-oriented programming language (OOPL); Attribute-oriented Programming (AtOP)

1. Introduction

The general goal of domain-driven design (DDD) [1, 2] is to develop software (iteratively) around a realistic model of the application domain, which both thoroughly captures the domain requirements and is technically feasible for implementation. According to Evans [1], object-oriented programming languages (OOPLs), such as Java [3, 4], are argued to be a natural fit for use with DDD. This is because objects both represent concepts in the domain model and are implemented directly in the code. In this paper, we will use DDD to refer specifically to object-oriented DDD.

The primary idea of DDD [1] is domain modelling for software development. Recently, we observe, based on our detailed analysis of the foundational works [1, 5] and the state-of-the-art DDD frameworks [6, 7], that a current trend in DDD is to utilise the domain model for software generation. Given a well-designed domain model, other parts of software, including graphical user interface (GUI) and the data source models can be generated automatically from it. Although DDD and model-driven software engineering (MDSE) arguably have in common the software generation aim [8], DDD follows a more agile approach to domain modelling. This approach relies on modern advance in OOPLs.

*Corresponding author at: Department of Software Engineering, VNU University of Engineering and Technology, Vietnam.
Email addresses: duclm@hanu.edu.vn (Duc Minh Le), hanhdd@vnu.edu.vn (Duc-Hanh Dang), hanv@vnu.edu.vn (Viet-Ha Nguyen)

Domain modelling is concerned with building a domain model for each subject area of interest of a domain. DDD considers domain model to be the core (or "heart" [1]) of software, which is where the complexity lies. DDD has two key tenets [1, 2]: (1 - *feasibility*) a domain model is the code and vice versa, and (2 - *satisfiability*) the domain model satisfies the domain requirements that are captured in a shared language, called the *ubiquitous language*. This language is developed by both the domain experts and the technical members of the project team as part of an iterative and agile process of investigating and analysing the domain requirements.

These two tenets, in our view, are the motivation behind the main research thread in DDD that had led to the development of contemporary object-oriented DDD software frameworks (Apache-ISIS [6] (successor of Naked Objects [5]) and OpenXava [7]). The modelling language is most commonly constructed from an annotation-based extension of OOPL.

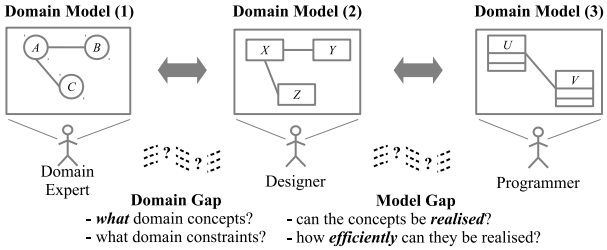

Figure 1: Domain model gaps and challenges.

A deeper research into the relevant literature reveals three reasons why this annotation-based extension is used in DDD. First, an annotation-based extension forms the basis for an **annotation-based DSL** [9] (*a.k.a* fragmentary, internal DSL [10]), and thus designing the domain model in it helps determine the implementation feasibility of the domain model in a target OOPL. An external DSL [10, 11], on the other hand, requires a operational semantic specification. Ensuring the correctness of this specification is a non-trivial task. Second, a domain model specified in an internal DSL can be treated as a program in the target OOPL. This allows the development team to leverage the capabilities of the language platform to not only process the model but to develop the software generator. This helps significantly reduce the development effort, compared to using an external DSL. Third, the practice of using annotations has been around for quite some time, especially in attribute-oriented programming (AtOP) [12–15], and proved useful. AtOP extends a conventional program with a set of annotation-based attributes which are used to define application- or domain-specific semantics. It is shown in [14], with empirical evidence, that the practice of recording the programmer's intention of the program elements using annotations is rather natural, since programmers' mental models do overlap. The use of purposeful annotations in programs help not only increase program correctness and readability [14] but raise its level of abstraction [15].

It can be gathered from [1] that an annotation-based DSL for DDD needs to overcome the challenges that arise from bridging two technical gaps. These gaps, which we depict in Figure 1, exist between the perceived domain models of three key stake holders in the DDD development team: domain expert, designer, and programmer. The *domain gap* is caused by mismatches in the views of the domain expert and designer in terms of what domain concepts and constraints need to be defined in the domain model. The *model gap* is caused by mismatches in the views of the designer and the programmer in terms of the implementation feasibility and efficiency of the domain model.

We argue that, on the one hand, the domain gap can be bridged by ensuring that the annotation-based DSL is both expressive and abstract, so that it can sufficiently and comprehensively express the domain. On the other hand, the model gap can be bridged by ensuring that the language is defined with respect to a high-level modelling language. Indeed, high-level modelling language (e.g. UML [16]) has been suggested in [1, 2] as being useful for analysing the domain requirements.

Existing work in DDD have four main limitations with respect to filling these gaps and software generation. First, their structural modelling solutions do not fully support association, do not formally define the essential structural constraints, and lack support for the structural correspondences between the domain state and behaviour spaces. We consider the domain state space of a domain class as a structure consisting of all the annotations attached to the class and its members that are concerned with the valid object states at any given point in time. The behaviour space of a domain class is a structure consisting of all the annotations attached to the class's methods that are concerned with the allowable changes in the object state. Second, existing work in DDD only partially support behavioural modelling[1] through the use of method-based action in domain class. Third, they do not define their extensions with respect to a high-level modelling language. Fourth, they do not precisely characterise the software that is generated from the domain model.

In this paper, we consolidate and extend previous work [17, 18] to address these limitations. We propose a **DSL-based domain-driven design method** (**L3D**) which consists in four parts (contributions):

1. An annotation-based DSL named **domain class specification language** (DCSL) for designing the domain model, the heart of which is the domain class and its structure.

2. A structural mapping between the state and behaviour spaces of a domain class. This mapping enables a technique for generating the behavioural specification.

3. A technique that uses DCSL to support behavioural modelling with UML activity diagram.

4. A 4-property characterisation of the software generated from the domain model.
   We demonstrate our method with a Java software tool and evaluate DCSL in the context of DDD.

As far as domain modelling is concerned, our method views domain class and its structure as forming the core of domain model. DCSL consists of an annotation-based extension that expresses a set of essential structural constraints and the essential behaviour of a domain class. The abstract syntax model of DCSL is defined using UML/OCL class diagram. We observe that DCSL enables a technique to support the behavioural modelling aspect through UML activity diagram. In essence, this involves identifying the core activity diagram patterns and modelling these into the domain model using an association scheme that mimics the activity flows. This model is then used to refine the domain classes with new structural and behavioural features.

With regards to software generation, we take a step back to systematically investigate how generativity can be achieved in developing not only the software but the domain model itself. For domain modelling, we propose a technique that uses a structural mapping between the modelling elements to automatically generate the behavioural specification of domain class. For software generation, we propose a DCSL-based, 4-property characterisation of the software that is automatically generated from the domain model. The properties include instance-based GUI, model reflectivity, modularity and generativity. We discuss, in particular, how generativity is achieved at three levels: view, module and software. We consider module as a self-contained software unit [19] that consists of a domain class, a view that renders objects of this class on a user interface, and a set of actions that are performed on this view. We demonstrate the applicability of our method using a Java software tool that we have developed. The tool takes a domain model as input and generates a software as output.

The rest of the paper is structured as follow. Section 2 explains in detail the motivation of our work and a running example. Section 3 gives an overview of our method. Section 4 defines DCSL. Section 5 discusses how DCSL is used to support behavioural modelling. Section 6 presents software generation and characterisation. Section 7 discusses our evaluation of DCSL in DDD's context. Section 8 reviews the related work. Finally, Section 9 gives the concluding remarks and states the future work.

[1] One of the two core modelling aspects supported by UML [16], which models how the system state is changed over time.

2. Background and motivation

This section explains in detail our motivation using a running a example. The example is a course management software domain (*abbr.* CourseMan). It describes a compact, yet complete, software domain that includes both structural and behavioural aspects. Figure 2 shows a partially completed CourseMan domain model expressed in the form of a UML class diagram.

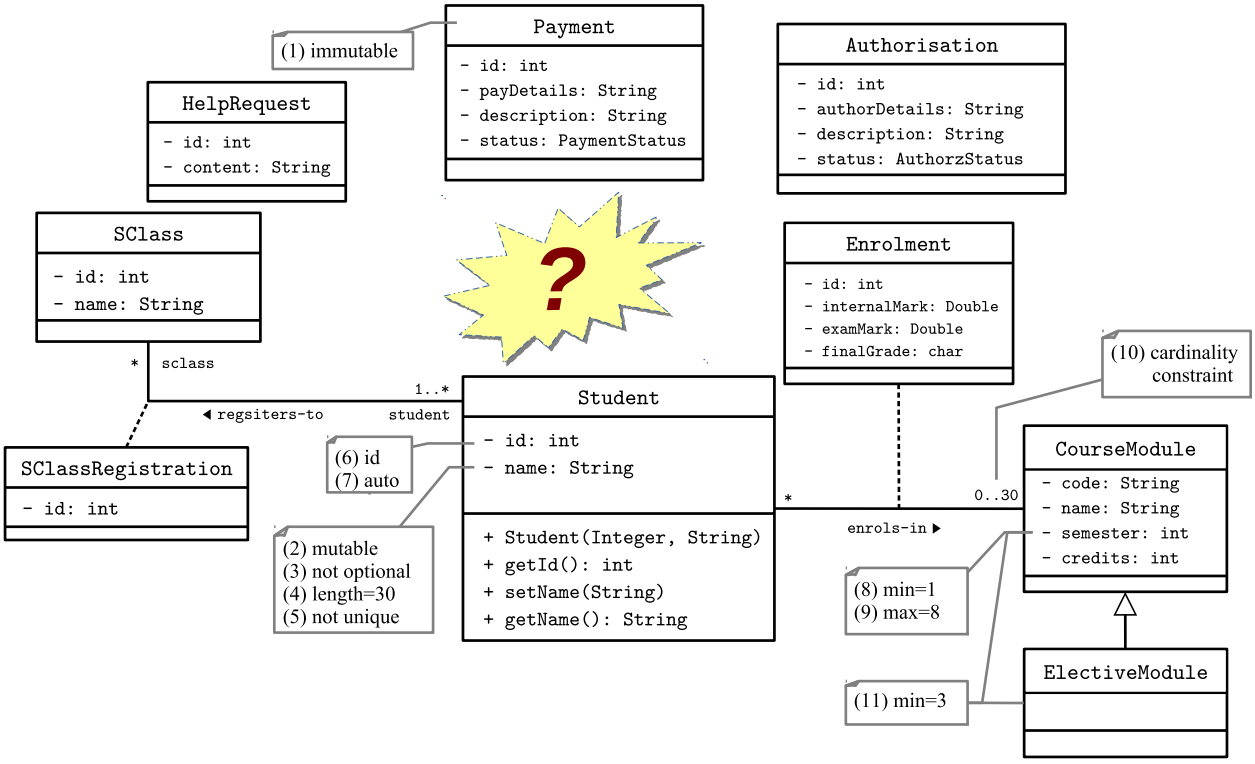

Figure 2: A partial domain model of CourseMan.

The bottom part of the figure shows four classes and two association classes of CourseMan. Class `Student` represents students that register to study in an academic institution. Class `CourseModule` represents the course modules that are offered by the institution. Class `ElectiveModule` represents a specialised type of `CourseModule`. Class `SClass` represents the student class type for students to choose. Association class `SClassRegistration` captures details about the many-many association between `Student` and `SClass`. Finally, association class `Enrolment` captures details about the many-many association between `Student` and `CourseModule`. The top part of Figure 2 (the area containing a star-like shape labelled "?") shows three other classes that are intended to capture the design of an enrolment management activity. Suppose that we know some design details (the attributes shown in the figure) and the following description about these classes:

- `HelpRequest`: captures data about help information provided to students.
- `Payment`: captures data about payment for the intuition fee that a student needs to make.
- `Authorisation`: captures data about the decision made by an enrolment officer concerning whether or not to allow a student to undertake the registered course modules.

In this work, we aim to tackle the following two inter-related DDD problems:

1. How do we use an annotation-based DSL to construct a domain model that expresses both the structural and behavioural aspects of the domain?
2. How do we characterise the software that is generated from such a domain model?

Before explaining these questions in detail, let us review the basic concepts concerning OOPL.

2.1. Background on OOPL

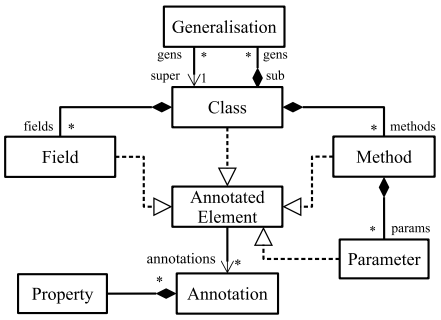

Figure 3: A high-level UML-based abstract syntax model of OOPL.

We adopt the metamodelling approach [11] to defining OOPL. Figure 3 shows a high-level UML-based **abstract syntax metamodel** (**ASM**) of OOPL. The ASM is high-level in the sense that it excludes the metaconcepts that describe the structure of the method body. Our primary interest in this paper is on the class structure, not the code. The **concrete syntax metamodel** is a textual syntax that is described in [4] for Java and in [20] for C#. The semantic metamodels of Java and C# are described also in [4] and [20] (*resp.*).

Note, in particular, the following three design features of the OOPL ASM shown in the figure. First, metaconcept Class represents both class and interface (an interface is a class that has no fields). Second, metaconcept AnnotatedElement is modelled as supertype of the four non-annotation metaconcepts (namely Class, Field, Method, and Parameter). The association between AnnotatedElement and Annotation helps capture a general rule that Annotation can be attached to any of the non-annotation metaconcept. Third, metaconcept Generalisation (adapted from UML's generalisation [16]) captures the subclass-superclass relationship between classes. Basically, a class may consist of one or more generalisations, each of which must be related to exactly one superclass.

Definition 1. An OOPL model (or **model** for short) is a structure that conforms to the metamodel of OOPL. The conforms-to relationship is as defined in [11]. Denote by $e : T$ a **model element** (or instance) e of some metaconcept T. □

In this paper, when it is necessary to highlight the metaconcept of model element we will use a/an/the with name of the metaconcept to refer to a particular element, and the plural form of this name to refer to a collection of elements. For example, a Java model for expressing the CourseMan's domain model in Figure 2 would include four Classes[2] : `Student`, `CourseModule`, `Enrolment`, `SClass` and `SClassRegistration`. These classes are instances of the metaconcept Class of Java.

Metaconcept Annotation borrows its name from Java [3, 4]. The equivalent metaconcept in C# is attribute [20]. Although annotation is syntactically defined as a Java interface it actually behaves like a class. This same view is also held in C#. Thus, in this paper we will consider annotation as being behaviourally equivalent to class. Because Annotation plays a key role in the definition of DCSL (discussed

[2]In this paper, we use normal font for the core and built-in meta-concepts, and `fixed font` for domain-specific meta-concepts.

in Section 4), in the remainder of this section we will present a number of basic concepts concerning this metaconcept. Our objective is to promote Annotation to be a generic metaconcept that is applicable to both Java and C#.

Definition 2. Annotation (short for annotation type) is a metaconcept behaviourally equivalent to Class, that is designed for the purpose of being applied to other metaconcepts to present information about them in a uniform and structured fashion. The annotation instances are created at run-time for each model element to whose metaconcept the annotation is applied.

The application of annotation is also intuitively called **annotation attachment**. An annotation A may be attached to more than one non-annotation metaconcepts $T = \{T_1, ..., T_n\}$. We write this as A^T. □

When the context either does not require to identify or is clear as to what T in A^T is, we will omit it and simply write A^T as A. For example, to model domain classes (such as `Student` of CourseMan), we would introduce an annotation named `DClass` (discussed in detail in Section 4). This annotation is attached to Class, and so we write: $\texttt{DClass}^{\text{Class}}$.

An annotation consists of properties. A property is called an "annotation element" in Java and a parameter in C#.

Definition 3. An **annotation property** is characterised by name, a data type, and a default value. Data type is one of the following: primitive type, string, enum, class, annotation or an array of these types. We call a property an **annotation-typed property** if its data type is defined from annotation (i.e. is an annotation or an array thereof). □

The concept of annotation property differs slightly between Java and C#. In C#, an annotation-typed property cannot take an Annotation as its data type. Fortunately, this difference can be resolved syntactically using a transformation. We explain this transformation in detail in the extended version of this paper [21]. In essence, it involves "promoting" an annotation-typed property to an annotation that is attached directly to the target metaconcept. Thus, in this paper, we will use the Java's interpretation of annotation property (presented in Definition 3 above), because it helps bring a modular structure to annotation.

For example, to model a fact that objects of a domain class are mutable, we introduce into the annotation `DClass` a property named `mutable`. We write this property using the dot notation as `DClass.mutable`.

It follows from Definition 3 that there exists a reference relationship between the annotations. We say that an annotation A *directly references* an annotation B if there exists one annotation-typed property of A such that the data type of this property is defined from B. Further, we say that annotation A *indirectly references* annotation B if there exists another annotation C such that: A directly or indirectly references C and C directly references B. In both cases, A is the *referencing annotation* (of B) and B is the *referenced annotation* (by A). The next definition establishes two fundamental annotation property constraints.

Definition 4. An annotation property is **constrained** by the following:

1. if its data type is an annotation (or an array thereof), this annotation (or the annotation of the array element type) is neither the same as nor a referencing annotation of the annotation that owns the property.
2. the property value is a constant or an annotation element expression.

□

Definition 5. Given a model M and an annotation A^T, an **annotation element** a of A in M is an instance of A that is assigned to an element $e : T_i \in M$ (for some $T_i \in T$). We say that a is created by **assigning** A^{T_i} to e, and write $A^{T_i}(e) = a$. Conversely, we say that the model element e is *assigned with* an element of A, or e is a target element of A. □

When we are only interested in whether or not $A(e)$ is defined, we write $\texttt{def}(A(e))$ and $\texttt{undef}(A(e))$, *resp.*. For example, assigning `DClass` to `Student` results in the annotation element `DClass(Student)` being created. This element has property `mutable` being set to `true`. In Java, we write the assignment in textual form as follows:

```
@DClass(mutable=true)
public class Student {}
```

2.2. Structural modelling

In object-oriented modelling, the structural aspect is often modelled as a UML class model. Existing DDD frameworks [6, 7] model this aspect using an OOPL extension. However, their extensions have limitations which we discuss in the following subsections. The first objective of our work is to define an annotation-based DSL to address these limitations.

2.2.1. Representing association

Traditionally, an association is realised in OOPL as a set of fields, each of which realises an association end [22]. A field is own by a class that participates at an association end. A problem arises as to how to represent association in an annotation-based DSL? Existing DDD work either do not capture association [6] or only partially capture it [7].

2.2.2. Structural constraints

The UML note boxes in Figure 2 shows examples of 11 kinds of constraints concerning class, field and association. These constraints appear frequently in real-world domain models [16, 23–25]. We describe these constraints below:

1. Class `Payment` is immutable, i.e. all objects of this class are immutable.
2. Field `Student.name` is mutable, i.e. its value can be changed.
3. `Student.name` is not optional, i.e. its value must be specified for each object.
4. `Student.name` does not exceed 30 characters in length.
5. `Student.name` is not unique, i.e. its value may duplicate among objects.
6. `Student.id` is an id field.
7. `Student.id` is auto, i.e. its value is automatically generated.
8. The minimum value of `CourseModule.semester` is 1.
9. The maximum value of `CourseModule.semester` is 8.
10. The association constraint, e.g. with respect to the enrols-in association, a `Student` is enrolled in zero or no more than 30 `CourseModule`s, and a `CourseModule` is enrolled in by zero or more `Student`s.
11. The minimum value of `CourseModule.semester` in `ElectiveModule` is 3.

Our experience in DDD software development [17, 18, 26–29], and our study of the relevant literature had led us to a belief that there exists a set of *primitive constraints* for domain modelling, such as the ones listed above. These constraints are primitive in the sense that they are applied independently to a single model element (e.g. class `Student` and `Student.id`). Our aim is thus to define a *minimum (essential)* subset of constraints. To the best of our knowledge, existing work in DDD have not identified such a set.

2.2.3. Structural mapping and behaviour generation

Another objective of structural modelling is to define the essential methods that operate on the fields and their essential constraints. For example, class `Student` in Figure 2 illustrates three of the common types of operations: constructor, setter and getter. These are referred to more generally in [25] as creator, mutator and observer operations (*resp.*). A constructor, e.g. `Student`(Integer, String), is specified using a sub-set of the fields (e.g. `id` and `name`). A setter or getter is specified using the field, whose value it is operating on. For example, the two operations `Student.getName`(), `setName`(String) are specified using the field `Student.name`.

It is observed from these examples that there exist some form of *structural mapping* between a method and the field(s) whose values are manipulated by the method. If we can somehow identify this mapping then we can use it to devise a technique to systematically identify all the essential methods. Better still, if we can formalise the mapping then we can use them to improve design productivity by automatically generating the methods' behaviour.

However, existing DDD work have not yet investigated both structural mapping and behaviour generation.

2.3. Behavioural modelling

The structural mapping described above would provide a bridge between structural and behavioural modelling. In behavioural modelling using UML activity diagram, in particular, the behaviours captured in the behavioural model are gradually refined to a detailed form (called the *refined model*). The actions in this model are mappable to combinations of structural and behavioural features of the classes in the domain model. We argue that there are three key design questions concerning the refined model:

1. How do we define additional features for the domain classes that participate in the activity?
2. Given that it is beneficial to model the activity class (as described in [16]) directly in the model, how do we achieve this?
3. Is the annotation-based DSL sufficient for expressing the design requirements of the previous two questions?

These three questions are related. First, the activity class can be used as the base to connect the referenced domain classes and, through this, to identify the features that need to be added to these classes. Second, the sufficiency of the annotation-based DSL for expressing the design requirements can be examined in this process. To illustrate, let us consider the enrolment management activity of CourseMan that was introduced earlier. This activity involves performing a combination of the following high-level actions: student registration, student class registration, student enrolment into course modules and a number of support actions. The support actions include making intuition fee payment, authorising the enrolment details, and requesting for help.

A refined model of this activity would be mapped to some combination of the classes that are displayed in Figure 2. However, as indicated earlier there are missing design features in the model concerning how the domain classes in the top part of the figure are associated to the domain classes at the bottom part of the figure. These would be filled by modelling the activity class into the model and using it to bring these associations to the fore. This is achievable with the help of a software prototype (discussed in the next sub-section). The star-like area marked in the figure indicates the area that would be used for the activity class.

Hence, the next objective of our work is to investigate how behavioural modelling based on UML activity diagram can be supported.

2.4. Software generation and characterisation

It is understood that software generation can not be fully automated [30]. Nonetheless, the generally accepted position is that a GUI-based tool is employed to assist the development team in this process. In DDD, in particular, the generated software plays two important roles. First, it is used during domain model development to enable the development team to iteratively and interactively build the domain model. As discussed in [1], this phase is central to DDD. Second, the software is reused during production software development to quickly develop the production-quality software.

To accommodate these, we argue that the software be designed as a reusable prototype for the domain model. The last objective of our work is to characterise this software in terms of some key properties and to demonstrate these using a software generation tool that we have developed. Part of this objective is to investigate the role that our annotation-based DSL plays in generating the software.

3. Method overview

Before discussing the technical contributions of our work, let us explain how they fit in the context of our method. In L3D, domain modelling and software generation are intertwined and performed iteratively through a sequence of three steps (shown in Figure 4):

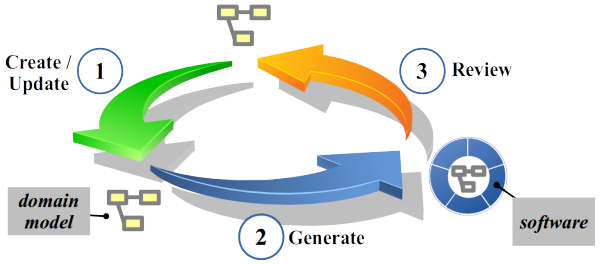

Figure 4: Method overview.

1. Create/Update domain model.
2. Generate software.
3. Review domain model (and the ubiquitous language).

To begin with, step 1 takes as input two initial high-level domain models that are typically (but not necessarily) expressed in UML class diagram and activity diagram and produce as output an initial domain model. The domain model is expressed in our proposed annotation-based DSL named DCSL. The initial model is usually incomplete and does not have enough design detail. The purpose of step 2 is to generate a software prototype that reflects the domain model in such a way that the development team can, in step 3, intuitively and collaboratively review the model (and the associated ubiquitous language). The next and subsequent cycles repeat at step 1 with an update that is performed on the domain model. The process is stopped when the domain model satisfies the requirements. From here, the domain model is used to develop the production software. The generated software prototype may be reused for this development.

Steps 1 and 2 are two key technical steps of our method. The technical contributions that will be discussed in the remainder of this paper provide a solution for these steps.

4. Domain class specification language (DCSL)

In this section, we present the design of our annotation-based DSL called **domain class specification language** (DCSL). As the name implies, DCSL is used for designing the domain class and its structure, which in our view form the core of domain model. This generic design makes DCSL a technical DSL [11], suitable for expressing different domain models. DCSL evolves from a Java specification language that we introduced in [27]. We first discuss the domain state space constraints that, together with a core sublanguage of OOPL, define the concepts that make up the domain of DCSL. After that, we specify DCSL as a language. Finally, we discuss the structural mapping that is induced by the DCSL's specification and how this gives rise to an automatic technique for generating the domain class behaviour.

4.1. Shaping the domain state space with constraints

The first ten structural constraint examples listed in Section 2 are in fact examples of 11 types of constraints that, in our view, are essential to the conception of domain class. Together, they help practically shape the state space of domain class, and thus make it suitable for domain modelling real-world software. Indeed, our study of the relevant system and software engineering literatures [16, 23–25] had led us to identifying these constraints. In the remainder of this paper, we will use the term **state space constraints** to refer to the constraints, and the term **domain state space**[3] (or **state space** for short) to refer to the state space restricted by them. Table 1 lists the names and natural language descriptions of the constraints. For the sake of exposition, we group the constraints into two types: (1) boolean constraint and (2) non-boolean constraint. The type of each constraint is indicated in the table.

In principle, the domain class behaviour consists in the operations that are performed on the domain state space. In the next section, we will discuss the essential operations that are specific for this state space.

4.2. Language specification

In essence, DCSL extends a sublanguage of OOPL with a set of *meta-attributes* for expressing the state space and the essential behaviour that operates on this state space.

Definition 6. A **meta-attribute** A^T is an annotation whose properties structurally describe the nonannotation metaconcepts T to which it is attached. □

[3] to differentiate from the term "system state space" used in formal methods.

Table 1: The essential state space constraints.

Constraints		Type	Descriptions
OM	Object mutability	Boolean	Whether or not the objects of a class are mutable [24, 25]
FM	Field mutability	Boolean	Whether or not a field is mutable (i.e. its value can be changed) [23]
FO	Field optionality	Boolean	Whether or not a field is optional (i.e. its value needs not be initialised when an object is created) [23]
FL	Field length	Non-Boolean	The maximum length (if applicable) of a field (i.e. the field's values must not exceed this length) [23]
FU	Field uniqueness	Boolean	Whether or not the values of a field are unique [23]
IF	Id field	Boolean	Whether or not a field is an object id field [23]
YV	Min value of a field	Non-Boolean	The min value (if applicable) of a field (i.e. the field's values must not be lower than it) [23]
XV	Max value of a field	Non-Boolean	The maximum value (if applicable) of a field (i.e. the field's values must not be higher than this) [23]
TF	Auto field	Boolean	Whether or not the values of a field are automatically generated (by the system) [23]
YL	Min number of linked objects	Non-Boolean	The minimum number of objects to which every object of a class can be linked [16]
XL	Max number of linked objects	Non-Boolean	The maximum number of objects to which every object of a class can be linked [16]

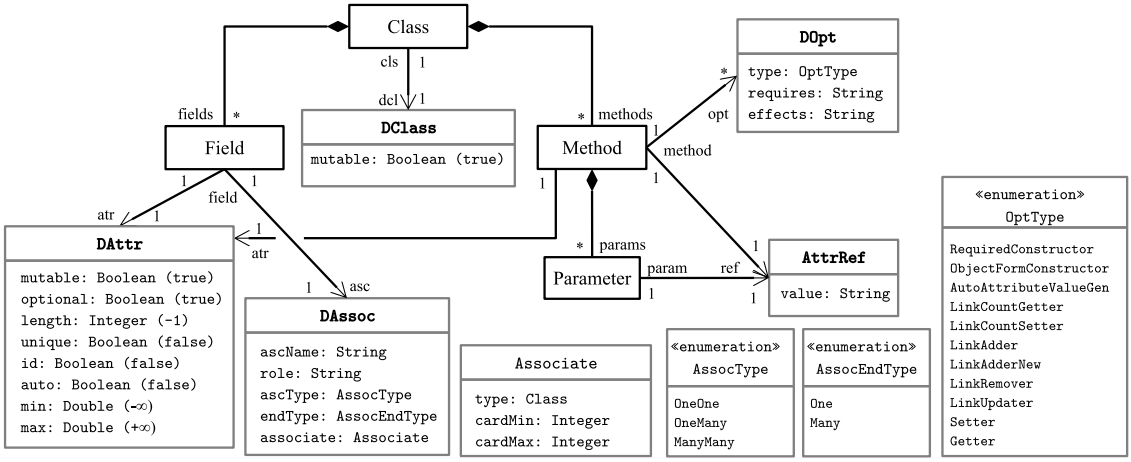

Figure 5: The abstract syntax model of DCSL.

Definition 7. DCSL is a language defined *w.r.t* an annotation-based extension $\langle \mathcal{T}, \mathcal{R} \rangle$ of a sublanguage L_O of an OOPL L as follow:

- L_O is a sub-language of L that consists of Class, Field, Method, Parameter and the relevant rules concerning these metaconcepts.
- $\mathcal{T} = \mathcal{A} \cup \mathcal{N}$, where:
  - $\mathcal{A} = \{\texttt{DClass}^{\{\text{Class}\}}, \texttt{DAttr}^{\{\text{Field}\}}, \texttt{DAssoc}^{\{\text{Field}\}}, \texttt{DOpt}^{\{\text{Method}\}}, \texttt{AttrRef}^{\{\text{Method, Parameter}\}}\}$
  - $\mathcal{N} = \{\texttt{Associate}, \texttt{AssocType}, \texttt{AssocEndType}, \texttt{OptType}\}$.
- $\mathcal{R}$ contains state space constraints and other well-formedness constraints that are modelled by $\mathcal{T}$.

We say that DCSL is an **annotation-based DSL** *w.r.t* the OOPL L. □

In our specification of DCSL, we will focus on the abstract syntax. The concrete syntax is assumed to be a subset of that of the host OOPL, which is applied to the meta-concepts in the abstract syntax. Figure 5 shows the UML-based ASM of DCSL. We will explain the constraints that make up the set $\mathcal{R}$ in Section 4.3. Graphically, an annotation is represented by a 2-compartment class box and a property by a class field. The default value of a property (if any) is written within brackets and appears after the property

type. The two meta-attributes `DClass` and `DAttr` together has properties that directly model the first nine state space constraints listed in Table 1. The other two constraints (YL and XL) are modelled by the third meta-attribute named `DAssoc`. The remaining two meta-attributes (`DOpt` and `AttrRef`) model the essential behaviour of Method.

In order to model YL and XL, we design `DAssoc` to capture the association link context within which these two constraints are applied. The base property is `DAssoc.associate` (type: `Associate`), which captures the associated class (property `Associate.type`), together with the range of the number of linked objects of this class. This range is specified by the two properties `Associate.cardMin` (for YL) and `cardMax` (for XL).

In addition to these, we also introduce a number of other contextual properties that are required to identify the different associations realised by the same class. These include property `DAssoc.ascName`, which specifies the association name, property `ascType` (type: `AssocType`, specifying three association types: `OneOne`, `OneMany`, and `ManyMany`), property `role` (specifies the role of the link end), and property `endType` (type: `AssocEndType`, specifying two link end types: `One` and `Many`).

Meta-attribute `DOpt` in Figure 5 is used to specify the method behaviour, while meta-attribute `AttrRef` is used to specify the reference to a field whose value is manipulated by a method. Property `DOpt.type` defines the behaviour type, which is taken from the values captured by the enum `OptType`. In order to arrive at the 11 essential `OptType`s shown in the figure, we took the three core operation types discussed in [25] (namely creator, mutator, and observer) and specialised these for our state space structure discussed above. In addition to the behaviour type, we introduce two properties `requires` and `effects` for defining the pre- and post-condition of a method. We will explain more about these properties after introducing an example.

4.2.1. Example: CourseMan

Listing 1: A partial specification of the domain class `Student`

```
1  @DClass(mutable=true)
2  public class Student {
3
4    /*** STATE SPACE **/
5    @DAttr(type=Type.Integer,id=true,auto=true,mutable=false,
6           optional=false)
7    private int id;
8
9    @DAttr(type=Type.String,mutable=true,optional=false,
10          length=30)
11   private String name;
12
13   @DAttr(type=Type.Collection)
14   @DAssoc(ascName="std-has-enrols",role="student",
15     ascType=AssocType.One2Many,endType=AssocEndType.One,
16     associate=@Associate(type=Enrolment.class,cardMin=0,cardMax=30))
17   private Collection<Enrolment> enrolments;
18
19   /*** BEHAVIOUR SPACE (partial) **/
20   @DOpt(type=DOpt.Type.RequiredConstructor,
21     requires="n.size() <= 30",
22     effects="self.id = genId() and self.name = n")
23   public Student(@AttrRef("name") String n);
24
25   @DOpt(type=DOpt.Type.AutoAttributeValueGen,
26     effects="Student::idCount = Student::idCount+1 and result = Student::idCount")
27   @AttrRef("id")
28   private static int genId();
29
30
31
```

```
32    @DOpt(type=DOpt.Type.Getter,
33      effects="result = self.name")
34    @AttrRef("name")
35    public String getName();
36  }  // end Student
```

Listing 1 shows a partial DCSL specification of the class `Student` of CourseMan, which is written in Java. A more complete specification segment is shown in Appendix B. We use this example to illustrate DCSL and explain a graphical notation for its specifications that we use in the remainder of this paper. As shown in the listing, class `Student` is specified with `DClass.mutable` = `true`, because at least one of its domain fields (`name`) is mutable. The domain field `name` is mutable (`DAttr.mutable` = `true`), not optional (`DAttr.optional` = `false`) and has max length (`DAttr.length`) = 30. The domain field `id` is an identifier field (`DAttr.id` = `true`), whose value is automatically generated (`auto` = `true`). The former also means that `id` is not optional, while the latter means that this field is immutable. Finally, the domain field `enrolments` is an associative field. Its `DAssoc` assignment specifies that the field realises the one end of a one-many association to the associate class `Enrolment` (`Associate.type` = `Enrolment`). The min and max cardinalities of the `Enrolment`'s end are 0 and 30, *resp.*.

Note that, compared to Listing 1, the listing in Appendix B includes an extra property `DAttr.name`, which is added in our implementation of DCSL to specify the name of domain field. This property does not add anything to `DAttr`. It is merely used as a syntactic convenience for mapping a `DAttr` to its attached field. This mapping helps ease reading and processing the specification.

The required constructor of class `Student` has `DOpt.type` = `RequiredConstructor` and one parameter `n`, which is assigned with an `AttrRef` that references the domain field `name`. This is the only non-collection typed, non-optional domain field of class `Student`. `AttrRef` is also assigned to the getter method `getName`, which has `DOpt.type` = `Getter`. This `AttrRef` specifies that this method is the getter for the field `name`.

To ease discussion in the paper, we use the graphical notation shown in Figure 6 to intuitively present a DCSL model and its specification. This notation extends the UML graphical notation with a structured text notation. Specifically, nonannotation elements are drawn using standard UML class diagram notation. The meta-attribute elements are drawn using UML note box. Meta-attribute assignment is represented by a dashed grey line, whose target element end is marked with the attachment symbol (■). The note box content takes the form A $\{props\}$, where A is the meta-attribute name and *props* is a property listing. Each entry specifies the initialisation of a property to a value. If this value is another annotation element then this element is written using a nested, borderless note box. The entries are separated by either a next line or a comma (',') character.

Another feature of the notation is the use of a virtual (dashed) association line to represent a pair of `DAssoc` elements that help realise the association ends of an association. This association line is more compact and thus helps significantly ease drawing and improves readability of the model. We will often use the term "association" to refer this association line and the DCSL model elements that realise it.

The DCSL model in Figure 6 realises the DCSL specification segment of CourseMan shown in Appendix B. This model focuses on two domain classes `Student` and `Enrolment` and the association between them (class `Enrolment` realises the association class between `Student` and `CourseModule`). The association named "std-has-enrols" between `Student` and `Enrolment` is represented in the figure by the association line that connects the two classes. The two association ends are realised by two associative fields: `Student.enrolments` (explained above) and `Enrolment.student`. The two thick arc arrows in the figure illustrate how the two `DAssoc` elements of these fields are intuitively "folded into" the association line. The association name is used to label the line and the properties `DAssoc.ascName` of the two elements. The role label of an association end is used to set the value for the property `role` of the `Associate` element of the `DAssoc` element of the same end. The role label is also used to name the field at the opposite end of the association. The cardinality constraint of an association end is used to determine the suitable values for the two properties `ascType` and `endType` of the `DAssoc` element of the same end. In addition, the constraint is used to set the values of the properties `cardMin` and `cardMax` of the `Associate` element of the `DAssoc` element of the opposite end.

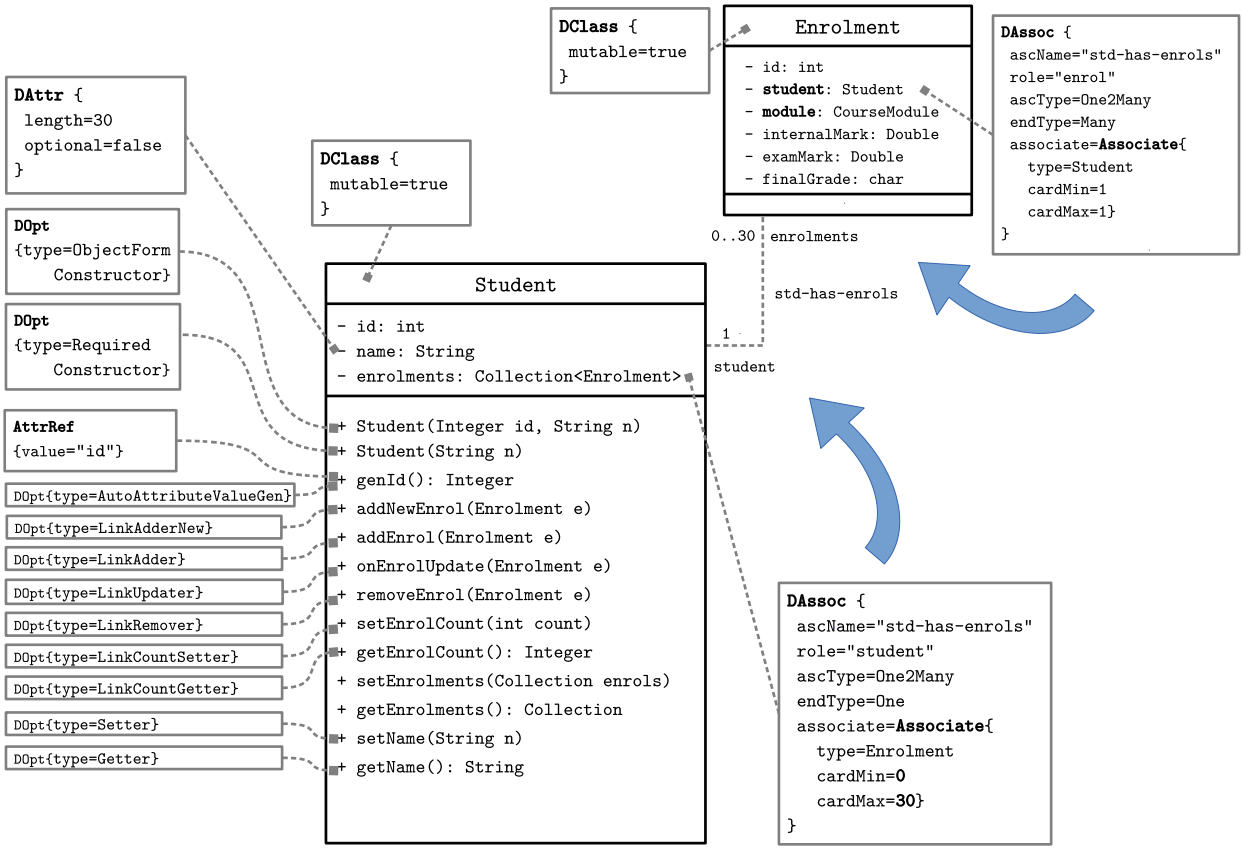

Figure 6: A DCSL model for a part of the CourseMan domain model.

4.2.2. Expressing the pre- and post-conditions of method

As discussed earlier, we use two properties `DOpt.requires` and `effects` for specifying the pre- and post-condition of method. The values of these properties are string-representations of OCL expressions. What we are interested in is not the use of general purpose OCL expression but a specific set of essential OCL expressions that are suitable for expressing the behaviour of the different `OptTypes`.

For example, Listing 1 shows how the pre-condition of the required constructor is the OCL expression `n.size()<= 30`, which states that the parameter `n` does not exceed 30 characters in length. This max length value is taken from the property `DAttr.length` of `Student.name`. Recall that this field is mapped to `n` via the specification of `AttrRef`. The post-condition of the constructor is the OCL expression `self.id = genId() and self.name = n`, which basically states that `Student.name` is set to `n` and `Student.id` is set to a value automatically generated by the helper method `genId`.

As another example, method `genId` is typed `AutoAttributeValueGen` and is linked to `id` via the specification of `AttrRef`. The OCL expression in the post-condition of this method basically states that the result of the method is an automatically incremented value, which is maintained by a static field `Student.idCount`.

4.2.3. Domain terms

The meta-attributes and their associations to the non-annotation metaconcepts provide basis for the definitions of four domain terms. We propose that these terms be added to the ubiquitous language's vocabulary.

Definition 8. Given a DCSL model M. An element c : Class $\in M$ is a **domain class** if c is assigned with a `DClass` element. An element f : Field $\in M$ is a **domain field** if f is assigned with a `DAttr` element. A domain field $f \in M$ is an **associative field** if f is assigned with a `DAssoc` element. An element m : Method $\in M$ is called a **domain method** if m is assigned with a `DOpt` element. □

Note that the definition of domain method only requires the assignment of `DOpt`, not `AttrRef`. We make this annotation optional to help ease the specifications of domain methods, especially when it can be inferred from the class structure which domain field is manipulated by a domain method. For example, one can easily infer the domain field of a setter/getter method.

4.3. Constraints on the ASM

We use OCL invariant to precisely define the constraints on the ASM. The benefit of using invariant is that it allows us to specify exactly the structural violation of the constraint, which occurs if and only if the invariant is evaluated to `false`. For the sake of exposition, we divide the OCL constraints into two groups: (1) state space constraints and (2) well-formedness constraints. State space constraints are those described earlier in Table 1. These constraints are identified with the same name labels as shown in this table. Well-formedness constraints are other OCL constraints needed to ensure the overall well-formedness of DCSL models. These constraints are identified with the name label prefix 'WF'.

The OCL definitions of the core ASM constraints are given in Table A.6 of Appendix A. The helper OCL functions that are used in the constraint statements are defined in the extended paper [21]. In the following paragraphs, we will explain how to interprete the two types of state space constraints listed in earlier in Table 1 and the generalisation constraint.

We define each constraint as an OCL invariant on an OOPL metaconcept. The invariant expression has the general form: ϕ `implies E`, where `E` is an OCL expression on the ASM structure that must *conditionally* be true in order for the constraint to be satisfied. The condition ϕ, upon which `E` is evaluated, is defined based on a certain characteristic of the metaconcept. In general, ϕ is `true` when the characteristic is in effect. If ϕ = `true` then `E` is evaluated and the result equates the constraint's satisfaction. Otherwise, `E` does not need to be evaluated (can be either `true` or `false`). Note that for state space constraint, in particular, ϕ is `true` when the corresponding property of the constraint is given a definite value.

4.3.1. Boolean constraints

Boolean constraints include OM, FM, FO, FU, IF, and TF. These constraints are expressed by the following Boolean-typed properties: `DClass.mutable` and `DAttr.mutable`, `optional`, `unique`, `id`, `auto`.

In Table A.6, a boolean constraint is an OCL expression of the form: `X.P implies E`, where ϕ = `X.P` denotes value of the boolean property `P` of some model element `X` (`X` can be a navigation sequence through the association links between model elements $e_1.e_2 \ldots e_n$). For example, let us consider the constraint FM:

```
1 -- exists a method p of a.owner s.t p's result expression is
2 --  a VarAssignExpression for a
3 context a: Field inv Mutable :
4   a.atr.mutable implies a.owner.methods->exists(p | p.isMutatorRef(a) and
5     let e : OclExpression = p.resultExp in
6     not(e.oclIsTypeOf(OclVoid)) and e.oclIsTypeOf(VarAssignExpression) and
7     e.oclAsType(VarAssignExpression).lhs.referredVariable = a)
```

In this constraint, the condition ϕ = `a.atr.mutable`, in which `X = a.atr` refers to the element `AttrRef(a)` of some Field `a` and `P = mutable` refers to the property `AttrRef.mutable`. The expression `E` is the OCL expression that immediately follows the keyword `implies`. This expression states the existence of a method `p` as described in the header comment.

4.3.2. Non-Boolean constraints

Non-boolean constraints include FL, YV, XV, YL, and XL. These are expressed by the following properties: `DAttr.length`, `min`, `max` and `Associate.cardMin`, `cardMax`.

In Table A.6, a non-boolean constraint is an OCL expression of the form: `X.P op v implies E`, where ϕ = `X.P op v` denotes an OCL expression that compares, using operator `op`, the value of some property `X.P` to some value `v`. For example, let us consider the constraint FL:

```
-- for all operation p of a.owner that mutates a via a parameter m,
--  exists an expression e in p's pre that restricts (m.size() <= l)
context a: Field inv MaxLength :
  let l : Integer = a.atr.length in
  l >= 1 implies
  if a.type.name = 'String' then
     a.owner.methods->forAll(p | p.isCreatorOrMutatorRef(a) implies
      p.params->exists(m | m.isAttrRef(a) and
        p.preExps().exists(e | e.equals(m.name+'.size() <= ' + l)) ))
  else
    true
  endif
```

In this constraint, the condition ϕ = `a.atr.length >= 1` (`X.P` = `a.atr.length`, `op` = '>=', and `v` = 1). `E` is the OCL expression that immediately follows the keyword `implies`. This expression states the universal truth about the operations `p` that is described in the header comment. Note that `E` is actually a disjunction of two expressions. The first expression, which is evaluated only when `a` is typed `String`, captures the meaning of `E` stated in the header comment. The second expression is `true`, and is evaluated when otherwise. In this case we do not care about `E` and thus must specify `true` to make the invariant hold.

4.3.3. Generalisation constraint

A key requirement of the ASM is that state space constraints are preserved through inheritance. To achieve this, in addition to the design rules enforced by OOPL we introduce a design rule on all the overridden methods that reference a domain field of an ancestor class in the inheritance hierarchy. Informally, this rule (WF4 in Appendix A) states that such an overridden method must be assigned with an `DAttr` that preserves the `DAttr` of the referenced domain field.

```
-- all overridden methods of a subtype that reference a domain field must preserve
-- the DAttr of that field
context c : Class inv:
  not(Tk::ancestors(c).oclIsUndefined()) implies
    c.overridenMethods()->forAll(p | not(p.atr.oclIsUndefined()) implies
      Tk::ancestors(c)->exists(c1 |
        c1.fields->exists(a | p.isAttrRef(a) and p.atr.preserves(a.atr)))
    )
```

In this constraint, the condition ϕ = `not(Tk::ancestors(c).oclIsUndefined())` means that the class c is a subclass of some superclass. For example, the following listing shows how the subclass `ElectiveModule` is specified with an overridden method named `getSemester`. This method has two annotations: (1) an `AttrRef` element that references the domain field `semester` of the superclass `CourseModule` and (2) a `DAttr` element that preserves the `DAttr` element of `CourseModule.semester`. The preservation is guaranteed by the fact that all properties except `min` have the same values, while the value of property `min` is 3, which is greater than `semester.min` = 1.

```
public class ElectiveModule extends CourseModule {
  @DOpt(type=DOpt.Type.Getter) @AttrRef("semester")
  @DAttr(type=Type.Integer,optional=false,min=3)
  @Override
  public int getSemester();
}
```

4.4. Structural mapping between state and behaviour spaces

The OCL definitions of the state space constraints imply that there exist structural correspondences between the domain fields and the domain methods that manipulate their values. For example, a mutable

domain field requires the existence of at least one domain method that can update its value. A more important observation, however, is that these correspondences can be captured and formalised using the DCSL's meta-attributes. In this section, we discuss a mapping formalism for the correspondences. In the next section, we propose a programatic technique that uses this mapping to generate a domain class behaviour from its state space.

Our approach is to analyse the meta-attribute assignments of a domain class and to extract from these a set of rules that map the state space to the behaviour space. We call these rules **structural mapping rules** and a set of these rules a **structural mapping**.

4.4.1. Mapping formalism

First, we use the meta-attribute assignments to define two design spaces. Let us denote by c_F, c_O the sets of domain fields and domain methods (*resp.*) of a domain class c. Further, denote by c_R the set of overridden methods of c (c_R may be empty).

Definition 9. The **state space** of a domain class c of a DCSL model M is the tuple $\langle \alpha, \mu_{\alpha,\mathtt{c}}, \phi_{\mathtt{c}} \rangle$, where $\alpha = \{\mathtt{DClass}, \mathtt{DAttr}, \mathtt{DAssoc}\}$, $\phi_c = \{c\} \cup c_F \cup c_R$ and $\mu_{\alpha,c} = \{(a, e) \mid A \in \alpha, e \in \phi_c, a = A(e)\}$. □

For example, the state space of the domain class `Student` that includes the association between `Student` and `Enrolment` is $\langle \alpha, \mu_{\alpha,\mathtt{Student}}, \phi_{\mathtt{Student}} \rangle$, where $\phi_{\mathtt{Student}} = \{\mathtt{Student}, \mathtt{id}, \mathtt{name}, \mathtt{enrolments}\}$ and $\mu_{\alpha,\mathtt{Student}} = \{(\mathtt{DClass}(\mathtt{Student}), \mathtt{Student}), (\mathtt{DAttr}(\mathtt{id}), \mathtt{id}), (\mathtt{DAttr}(\mathtt{name}), \mathtt{name}), (\mathtt{DAttr}(\mathtt{enrolments}), \mathtt{enrolments}), (\mathtt{DAssoc}(\mathtt{enrolments}), \mathtt{enrolments})\}$. The complete state space that additionally includes the association between `Student` and `SClassRegistration` is defined similarly, but will be omitted for the interest of space.

Definition 10. The **behaviour space** of a domain class c of a DCSL model M is the tuple $\langle \beta, \mu_{\beta,\mathtt{c}}, \lambda_{\mathtt{c}} \rangle$, where $\beta = \{\mathtt{DOpt}, \mathtt{AttrRef}\}$, $\lambda_c = c_O$ and $\mu_{\beta,c} = \{(b, e) \mid B \in \beta, e \in \lambda_c, b = B(e)\}$. □

To illustrate, let us consider a partial behaviour space of the class `Student` shown in Figure 6. We will discuss shortly how the complete behaviour space is identified and generated. The partial behaviour space $\langle \beta, \mu_{\beta,\mathtt{Student}}, \lambda_{\mathtt{Student}} \rangle$ has $\lambda_{\mathtt{Student}} = \{\mathtt{Student}[2], \mathtt{getName}\}$, and $\mu_{\beta,\mathtt{Student}} = \{(\mathtt{DOpt}(\mathtt{Student}[2]), \mathtt{Student}[2]), (\mathtt{AttrRef}(\mathtt{Student}[2].0), \mathtt{Student}[2].0), (\mathtt{AttrRef}(\mathtt{Student}[2].1), \mathtt{Student}[2].1), (\mathtt{DOpt}(\mathtt{getName}), \mathtt{getName})\}$. Here, `Student`[2] denotes the 2-parameter constructor `Student`(Integer, String), and `Student`[2].n ($n \in \{0, 1\}$) denotes the first and second parameters of the constructor.

A key observation that we make is that we can structurally map element patterns of the state space to those of the behaviour space.

Definition 11. Given a state space $\langle \alpha, \mu_{\alpha,\mathtt{c}}, \phi_{\mathtt{c}} \rangle$, k meta-attributes $A_1, \ldots, A_k \in \alpha$, and k stateful functions $u_i : A_i \to \{\mathtt{true}, \mathtt{false}\}$ ($i = 1 \ldots k$). A **state space element pattern** (**SSEP**) *w.r.t* (A_1, u_1) ,..., (A_k, u_k) is the set $\{(a, e) \mid (a, e) \in \mu_{\alpha,c}, a = A_1(e), u_1(a), \ldots, a = A_k(e), u_k(a)\}$. To ease notation, we will write SSEP by listing just the pairs (A_i, u_i). □

A stateful function is named after the property of the meta-attribute (A_i) and the condition to be applied to that property's value. We will discuss the set of all the essential stateful functions for DCSL in the next section. For example, function `isMutable` in the SSEP (`DAttr`, `isMutable`) means to check property `DAttr.mutable` for the condition that its value being set to `true`. The SSEP itself is thus the set of all the `DAttr` assignments of a state space in which `DAttr.mutable`=`true`. In the context of the `Student`'s state space given above, this set is $\{\mathtt{DAttr}(\mathtt{name}), \mathtt{DAttr}(\mathtt{enrolments})\}$.

For the sake of exposition, we next define a special type of SSEP which involves at least two meta-attributes, at least one of the stateful functions associated to which are the special stateful function named `unassign`. This function "reverses" the effect of assigning a meta-attribute to an element. That is, for any i: $(a = A_i(e) \wedge \mathtt{unassign}(a)) \leftrightarrow \mathtt{undef}(A_i(e))$.

Definition 12. Given a state space $\langle \alpha, \mu_{\alpha,\mathtt{c}}, \phi_{\mathtt{c}} \rangle$, k meta-attributes $A_1, \ldots, A_k \in \alpha$ ($k > 1$), k stateful functions $u_i : A_i \to \{\mathtt{true}, \mathtt{false}\}$ ($i = 1 \ldots k$), and $\exists i \in [1 \ldots k].u_i = \mathtt{unassign}$. An SSEP *w.r.t* (A_1, u_1),..., (A_k, u_k) is called an **SSEP with negation** (**SSEPN**). □

For example, (DAttr, isAuto), (DAssoc, unassign) is an SSEPN. It is the set of all the DAttr assignments of a state space in which DAttr.auto=true and the target elements are not assigned with any DAssocs. In the context of the Student's state space given above, this set is {DAttr(id)}.

Definition 13. Given a behaviour space $\langle \beta, \mu_{\beta,\mathtt{c}}, \lambda_{\mathtt{c}} \rangle$, k meta-attributes $B_1, \ldots, B_k \in \beta$, and k stateful functions $v_i : B_i \rightarrow \{\mathtt{true}, \mathtt{false}\}$ $(i = 1 \ldots k)$. A **behaviour space element pattern** (**BSEP**) *w.r.t* (B_1, v_1), $\ldots$, (B_k, v_k) is the set $\{(b, e) \mid (b, e) \in \mu_{\beta,c}, b = B_1(e), v_1(b), \ldots, b = B_k(e), v_k(b)\}$. To ease notation, we will write BSEP by listing just the pairs (B_i, v_i). □

For example, the BSEP (DOpt, isRequiredConstructorType) is the set of all the DOpt assignments of a behaviour space in which DOpt.type = RequiredConstructor. In the context of the Student's behaviour space in the example of Definition 10, this set is {DOpt(Student[2])}.

Now, let us denote by Σ and Π the sets of SSEPs and BSEPs of the state and behaviour spaces of a domain class (*resp.*).

Definition 14. Structural mapping from the state space $\langle \alpha, \mu_{\alpha,\mathtt{c}}, \phi_{\mathtt{c}} \rangle$ to the behaviour space $\langle \beta, \mu_{\beta,\mathtt{c}}, \lambda_{\mathtt{c}} \rangle$ is an injective function: $\Sigma \rightarrow \mathcal{P}(\Pi)$, that maps a SSEP of the state space to a subset of BSEPS of the behaviour space according to a set of rules.

We call a domain class c **behaviour essential** if c's structure conforms to this mapping. □

4.4.2. Structural mapping rules

Table 2: The core structural mapping rules.

No	SSEPs				BSEPs	
	DAttr		DAssoc		DOpt	
	Property	Stateful func.	Property	Stateful func.	Property	Stateful func.
1	auto	isNotAuto	–	–	type	isObjectFormConstructorType
	type	isNotCollectionType	–	–		
2	auto	isNotAuto	–	–	type	isRequiredConstructorType
	type	isNotCollectionType	–	–		
	optional	isNotOptional	–	–		
3	mutable	isMutable	–	–	type	isSetterType
4	auto	isAuto	–	unassign	type	isAutoAttributeValueGenType
5	type	isCollectionType	ascType	isOneManyAsc	type	isLinkAdderNewType
					type	isLinkAdderType
					type	isLinkUpdaterType
			endType	isOneEnd	type	isLinkRemoverType
					type	isLinkCountGetterType
					type	isLinkCountSetterType
6	type	isDomainType	ascType	isOneOneAsc	type	isLinkAdderNewType

Table 2 lists the structural mapping rules for the different OptTypes of DCSL. Each numbered row of the table defines one structural mapping rule. The column labelled "SSEPs" lists the SSEPs of each rule, and the column labelled "BSEPs" lists sets of BSEPs of the same rule. The SSEPs are related to two meta-attributes DAttr and DAssoc. The stateful functions associated to these meta-attributes in each pattern are listed under the sub-column labelled "Stateful func.". The sub-column "Property" lists the properties (if any) of each meta-attribute in a pattern, whose values are conditioned by the corresponding stateful function.

Except for the stateful function unassign, other stateful functions are primitive stateful functions. Each primitive stateful function is defined for a single property. For patterns that involve more than one property, their stateful functions are straight-forwardly constructed by logical-ANDing the primitive stateful functions of the concerned properties.

A primitive stateful function is named after a property and the condition that is applied to that property's value. In particular, the name of each BSEP's stateful function is an OptType value. For example, function isAuto means to check if DAttr.auto=true. Similarly, function isCollectionType (isNotCollectionType) means to check if DAttr.type is (*resp.* is not) a type of Collection. As another example, function isSetterType means to check if DOpt.type=Setter.

Let us explain how to read the mapping rules using four representative examples. The first two examples are rules 2 and 3. Rule 3 maps the SSEP (DAttr,isMutable) to the BSEP (DOpt,isSetterType).

Rule 2 maps the SSEP (DAttr, isNotAuto ∧ isNotCollectionType ∧ isNotOptional) to the BSEP (DOpt,isRequiredConstructorType). The SSEP involves three properties and its stateful function is constructed by logical-ANDing three primitive stateful functions.

The third example is rule 4. This rule maps the SSEPN (DAttr,isAuto), (DAssoc,unassign) to the BSEP (DOpt,isAutoAttributeValueGenType). The last example is rule 5. This rule maps one SSEP to six BSEPs. The SSEP is (DAttr,isCollectionType), (DAssoc,isOneManyAsc ∧ isOneEnd). The six BSEPs are: (DOpt,isLinkAdderNewType), (DOpt, isLinkAdderType), (DOpt,isLinkUpdaterType), (DOpt,isLinkRemoverType), (DOpt,isLinkCountGetterType), and (DOpt, isLinkCountSetterType).

4.5. Behaviour generation

Alg. 1 BSpaceGen

Input: c: a domain class whose state space is specified
Output: c is updated with domain methods

```
   // create constructors
 1 let F_U = {f | f ∈ c_F, ¬isAuto(DAttr(f)),
                ¬isCollectionType(DAttr(f))}
 2 let F_R = {f | f ∈ F_U, ¬isOptional(DAttr(f))}
 3 if F_U ≠ ∅ then
 4   create in c object-form-constructor c_1(u_1, ..., u_m)
     (u_j ∈ F_U) // rule 1
 5 if F_R ≠ ∅ then
 6   create/update in c required-constructor c_2(r_1, ..., r_p)
     (r_k ∈ F_R) // rule 2
   // create other methods
 7 for all f ∈ c_F do
 8   create in c getter for f
 9   if isMutable(DAttr(f)) then
10     create in c setter for f // rule 3
11   if def(DAssoc(f)) then
12     if isOneManyAsc(DAssoc(f)) ∧ isOneEnd(DAssoc(f)) then
13       create in c link-related methods for f // rule 5
14     else if isOneOneAsc(DAssoc(f)) then
15       create in c link-adder-new for f // rule 6
16   if isAuto(DAttr(f)) ∧ undef(DAssoc(f)) then
17     create in c auto-attribute-value-gen for f // rule 4
```

In this section, we discuss a programatic technique that uses the structural mapping to automatically generate the behaviour specification of a domain class. Our technique is captured by Alg. 1, which we call BSpaceGen. The algorithm takes as input a domain class (c) whose state space is specified in DCSL and automatically generates in c a set of domain method definitions. We assume that each method type has a header template that is available in the target OOPL. The templates are similar to those of the class `Student` in Figure 6. To ease reading, we name the methods after their types. For example, the required-constructor method at line 6 refers to the method whose type is `RequiredConstructor`. An implementation of Alg. 1 in Java, together with an explanation of how to run it with the CourseMan example, is given in [31].

Theorem 1. ***Behaviour Generation***
The input domain class updated by Alg 1 is behaviour essential. □

A brief proof of Theorem 1 is as follow. This proof is helped by the marker comments in the algorithm text concerning the mapping rules that are applied. The two sets F_U and F_R (created at lines 1-2) are sets of class fields that satisfy the SSEPs of the two constructor rules 1 and 2 (*resp.*). Each set forms the parameters of a corresponding constructor. The constructors are created at lines 3-4 and 5-6. The create/update statement at line 6 is to allow for the case that these sets are the same. That is, if $F_U = F_R$ then object-form-constructor (c_1) is the same as the required-constructor (c_2). In this case, c_2 is not created. Instead, c_1 is assigned with an additional `DOpt` whose type is `RequiredConstructor`.

The `for` loop at lines 7-17 realises other structural mapping rules (3–6), which are applied to the individual class fields. Rule 3 is applied at line 10 to create a setter for the domain field f if it is mutable. Rule 4 is applied at line 17 to create an auto-attribute-value-gen method for f if it is an auto, non-associative field. If f is an associative field then rules 5 and 6 are applied (at lines 13 and 15 *resp.*) to create the required link-related methods for the two cases of one-many and one-one associations (*resp.*). □

Theorem 2. ***Complexity***
The worst-case time complexity of Alg 1 is linear in number of domain fields of the input domain class. □

The proof of Theorem 2 is as follow. The dominant parts of Alg. 1 are the formations of the sets F_U (line 1) and F_S (line 2) and the `for` loop (lines 7-17). All three parts have the same worst-case time complexity $O(|c_F|)$, i.e. linear in the number of the domain fields of c. First, the formulation of F_U requires a loop over elements of c_F. This loop has $|c_F|$ as the max number of iterations. Second, the formulation of F_R requires a loop over the elements of F_U. Thus, this loop also has $|c_F|$ as its max number of iterations. Third, the `for` loop iterates over the elements of c_F and, in each iteration, performs some method creation statements. The max number of these statements is a constant which equals the number of the essential operation types that are applicable to the mapping rules 3–6. Hence, the `for` loop also has the complexity of $O(|c_F|)$. □

5. Behavioural modelling with DCSL

In this section, we explain how DCSL is used to specify the behavioural model of software directly in the target OOPL. This, together with the structural modelling aspect discussed in the previous section, provides the basis for a unified design method for software. We choose the UML activity language [16] for behavioural modelling, because this language has been shown to be domain-expert-friendly [32, 33]. We first explain our approach for using DCSL to realise an activity model. After that, we discuss how the approach is applied to a set of core activity model patterns.

5.1. Modelling approach

Our approach is based on three ideas. The first idea is to use associative field of DCSL to capture the associations among the behavioural modelling elements and the structural counterparts. It is these associations that "glue" together the two modelling aspects in the same unified domain model. The second idea is to use this *unified model* as the base to iteratively and interactively (together with the domain experts) capture, analyse and design both modelling aspects. A key enabler for this is a software tool (discussed in Section 6), which automatically generates a software prototype from a DCSL model. The third idea is to identify five basic activity flow patterns and tackle these patterns.

Specifically, we define the activity class and each control node of an activity model as domain class in the domain model and add associations among these classes and the data domain classes according to an *association scheme*. The data domain classes, or *data classes* for short, are domain classes in the domain model whose objects are manipulated by the action nodes of the activity. The association scheme mimics the activity flows among the activity nodes by using an association for each of the following class pairs:

- activity class and a *merge class* (the domain class representing the merge node).
- activity class and a *fork class* (the domain class representing the fork node).
- a merge (*resp.* fork) class and a data class of an action node connected to the merge (*resp.* fork) node.
- activity class and a data class of an action node that is not connected to a merge or fork node.

Note that the activity flows actually are performed as part of the activity graph execution. In this paper, we assume that the activity graph exists and focus only on the base domain model representation above. The graph execution semantics is as described in the UML specification [16]. We have in fact implemented such graph as part of the software tool discussed in Section 6. We use this tool to run the behavioural modelling examples presented later in this section.

Let us give an intuitive explanation of the approach, using a sequential activity model of the enrolment management activity of CourseMan. This model is shown in the bottom part of Figure 7, which we will

discuss shortly in the next section. The model involves performing action student registration (references (data class) `Student`) and then student class registration (references `SClassRegistration`). We construct this model as follow. First, we model the activity class as a domain class named `EnrolmentMgmt`. Next, we add two associations to connect `EnrolmentMgmt` to `Student` and `SClassRegistration`. We then input this model into a software tool to generate a software prototype. The development team uses this software to observe, discuss, and update the two referenced domain classes with new features (if any). Further, they can also discuss the activity flow logic among the two actions. For example, the team would conclude that a weak dependency association from `SClassRegistration` to `Student` is all that is needed in order to display `Student` information on the registration form. They would also decide to add new domain fields (and methods) to each of these classes. The development team may also discuss the object flow from student registration to student class registration, and decide that this flow needs to include a data filter for just `Student.name` (all other `Student`'s fields are excluded).

5.2. Expressing the core activity modelling patterns

We now discuss how our approach is applied to the core activity modelling patterns. There are five patterns, which we name after the following primitive activity flows: sequential, decisional, forked, joined, and merged. This patterns set significantly extends and completes the result of our previous work [18], which only presents one behavioural modelling pattern. Due to space constraint, we will focus in this paper on the first three patterns. The other two patterns have similar designs to decisional and forked (*resp.*). We discuss these in the extended version of this paper [21].

We focus in particular on the design of the *pattern form* [34, 35]. To keep the patterns generic, we present for each pattern form the activity model and the **template domain model** that realises the activity model. The template model is a 'parameterised' DCSL model, in which elements of the nonannotation metaconcepts are named after the generic roles that they play. We use the DCSL model notation of Figure 6 to draw the pattern form. For brevity, we will assume that all fields that are listed in a DCSL model diagram are domain fields, and thus will omit from this diagram the `DAttr` assignments of these fields. Further, we will omit the base domain methods of each domain class.

Each pattern is illustrated with an example, which is a version of the enrolment management activity of CourseMan. A pattern example includes a concrete DCSL domain model and one or more software GUIs. The software is automatically generated from the domain model, using the software tool described in Section 6.

5.2.1. Sequential pattern form

We begin in this section with the design of the sequential pattern form. The top-right of Figure 7 shows the activity model, the top-left shows the template domain model. The latter consists of three classes `Ca`, `Cs`, and `Cn` and three associations. Class `Ca` is the activity class and has two associations with the two data classes `Cs` and `Cn`. The objects of these classes are manipulated by the action nodes e_s and e_n (*resp.*) of the activity model.

There are two one-many associations between `Ca` and `Cs` and `Cn`, and one one-many association between `Cs` and `Cn`. The first two associations are used to connect the activity to the two data classes. The third association is a weak dependency from `Cn` to `Cs`. It is added in order to realise the object flow from action e_s to action e_n.

Example

The remainder of Figure 7 shows how the pattern is applied to the enrolment management activity of CourseMan. The UML activity model is shown at the bottom left. This activity involves performing two actions in sequence. The first action (e_s) registers a student into course modules, while the second action (e_n) registers the student into a preferred class.

In this example, `Ca` = `EnrolmentMgmt`, `Cs` = `Student`, $n = 1$, `C1` = `SClassRegistration`. Note that the weak dependency association between `SClassRegistration` and `Student` is superseded by the existing association (shown in Figure 2) between these two classes. The GUI of the domain model is shown in Figure 8. It presents how the activity's GUI (`EnrolmentMgmt`'s in this example) contains the GUIs of the two actions in separate tabs. This provides the user with an awareness of the activity context, while

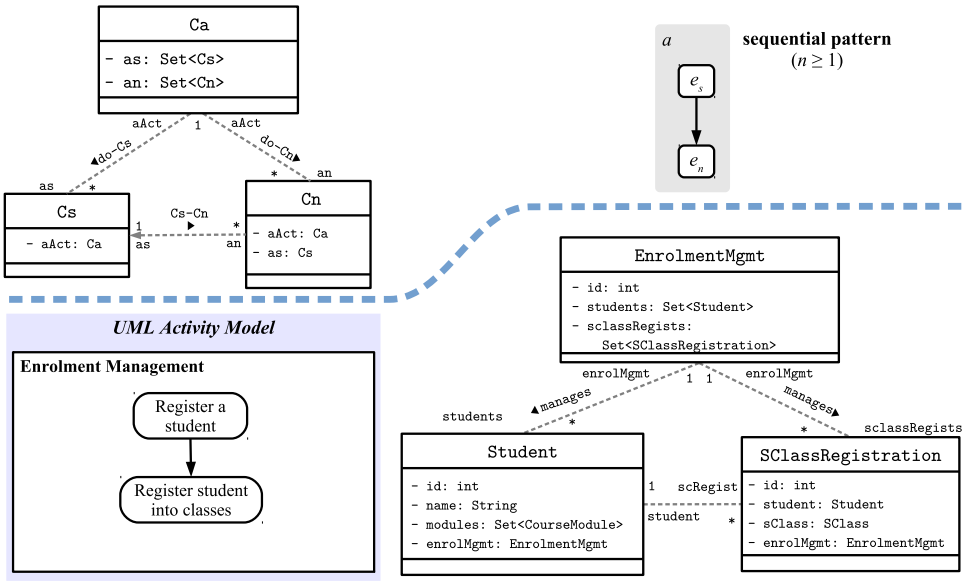

Figure 7: The sequential pattern form (top left) and an application to the enrolment management activity.

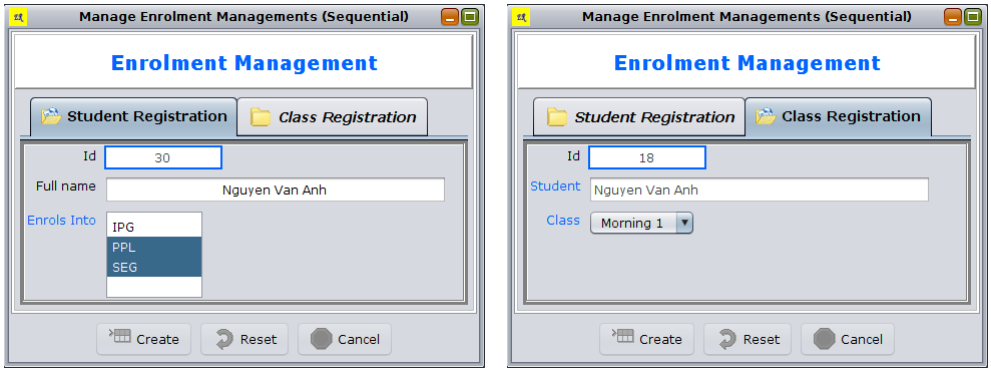

Figure 8: The sequential pattern form view of enrolment management activity.

performing a given action. The LHS GUI shows the tab containing the `Student`'s GUI, while the RHS GUI shows the tab containing the `SClassRegistration`'s GUI. Note, in particular, that the data field of the associative field `SClassRegistration.student` is automatically set to the value `Student`(`name`="Nguyen Van Anh"). This `Student` object is created on the `Student`'s GUI.

5.2.2. Decisional pattern form

The top-left of Figure 9 shows the template model that realises the activity model shown on the top-right. Apart from the activity class `Ca`, the template model includes five other domain classes, namely `Cd`, `D`, `C1`, `Cn`, and `Ck`, that are mapped to the five activity nodes. Class `Ck` is a *control class* that is referenced by the control node c_k of the activity model. Class `D` is a *decision class*, which is a domain class that is referenced by the decision node. This class implements an interface named `Decision`. The implementation is domain-specific and involves writing the control logic for the decision node. This logic requires knowledge of and may cause new features to be added to the three domain classes involved, namely `C1`, `Cn`, and `Ck`. Thus, we add three weak dependency associations to the model to associate class `D` to these three classes.

Class `Ca` has one-many associations to the other four domain classes. The configurations of these associations are similar to those in the sequential pattern's form. Note, in particular, that the association

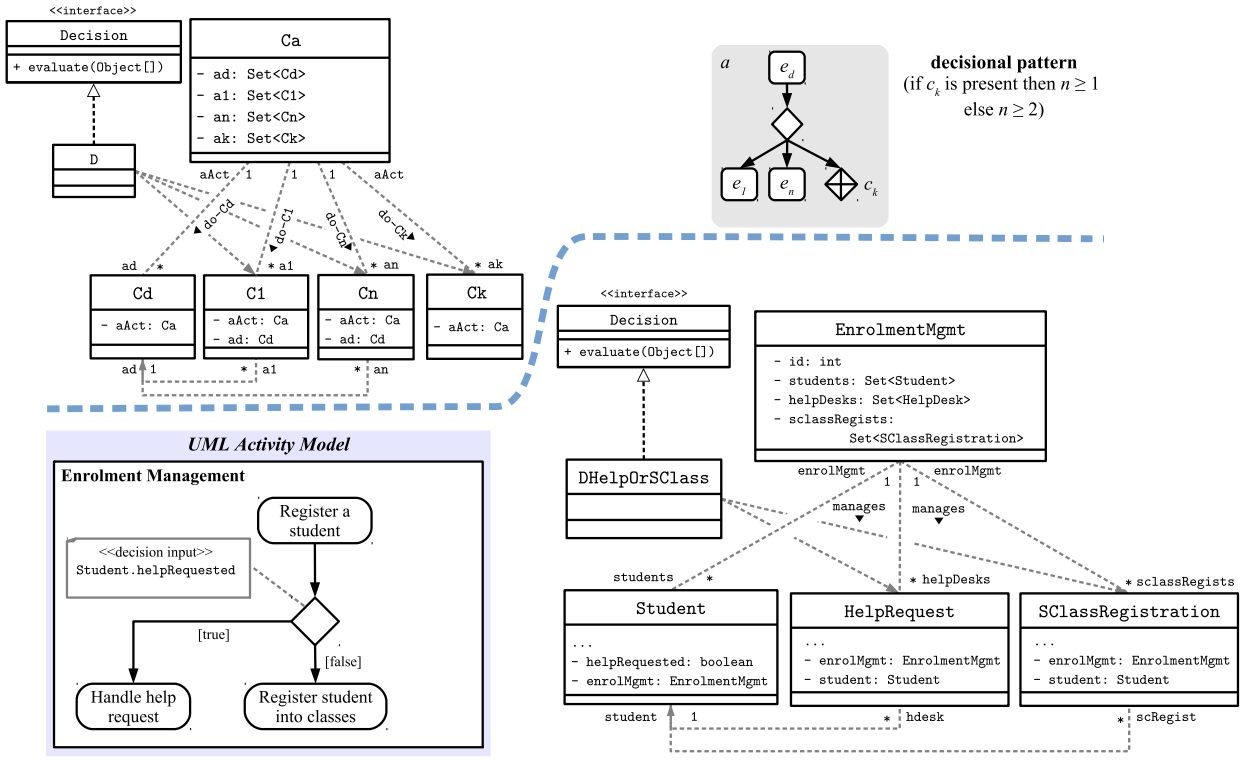

Figure 9: The decisional pattern form (top left) and an application to the enrolment management activity.

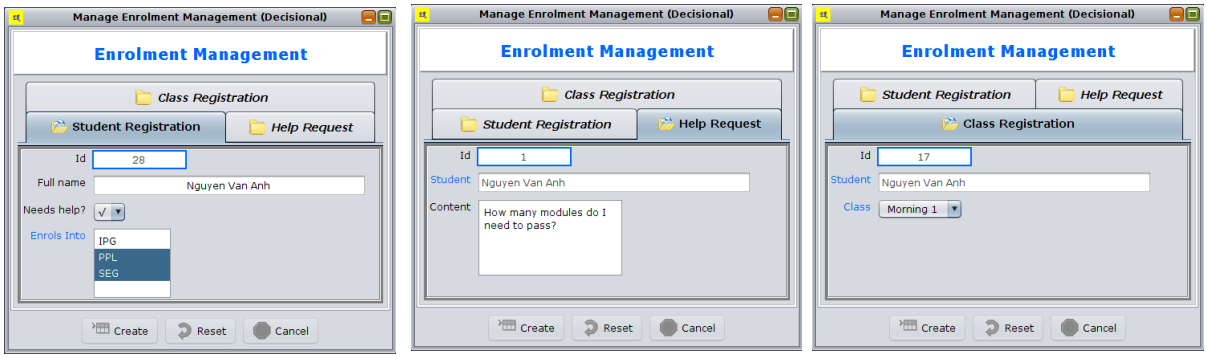

Figure 10: The decisional pattern form view of enrolment management activity.

to `Ck` can be used as a bridge in a larger activity model to other activity flow blocks. This association is applied differently if c_k is a decision node. In this case, the association to `Ck` is replaced by (or "unfolded" into) a set of associations that connect `Ca` directly to the domain classes of the domain model containing `Ck`.

The two weak dependency associations between `C1`, `Cn` and `Cd` reflect the fact that both `C1` and `Cn` know about `Cd`, due to the passing of object tokens from e_d to e_1 and e_n (via the decision node).

Example

The remainder of Figure 9 shows how the pattern is applied to the enrolment management activity of CourseMan. The UML activity model of this activity is displayed at the bottom left. In this activity, after performing student registration (e_d), a student may either (a) request help from the help desk (e_1) or (b) register into classes (e_2). The control node c_k is not used.

In this example, `Ca` = `EnrolmentMgmt`, `Cd` = `Student`, `D` = `DHelpOrSClass`, $n = 2$, `C1` = `HelpRequest`, C2 = `SClassRegistration`. The decision logic captured by `DHelpOrSClass` requires that a new boolean-typed domain field `Student.helpRequested` be added. If this field is set to `true` then the decision outcome is (a), otherwise the decision is (b).

Similar to the GUI of the sequential pattern, the activity's GUI (shown in Figure 10) contains the GUIs of the three actions in separate tabs. Under both cases of the decision, the `Student` object that is created in the first action (e.g. `Student(name`="Nguyen Van Anh")) is passed on to the next action. This object is then presented in the data field of the associative field `student` of the domain class mapped to this action. In Figure 10, the first GUI is for student registration. The second and third GUIs are for the cases that help request is and is not requested (*resp.*).

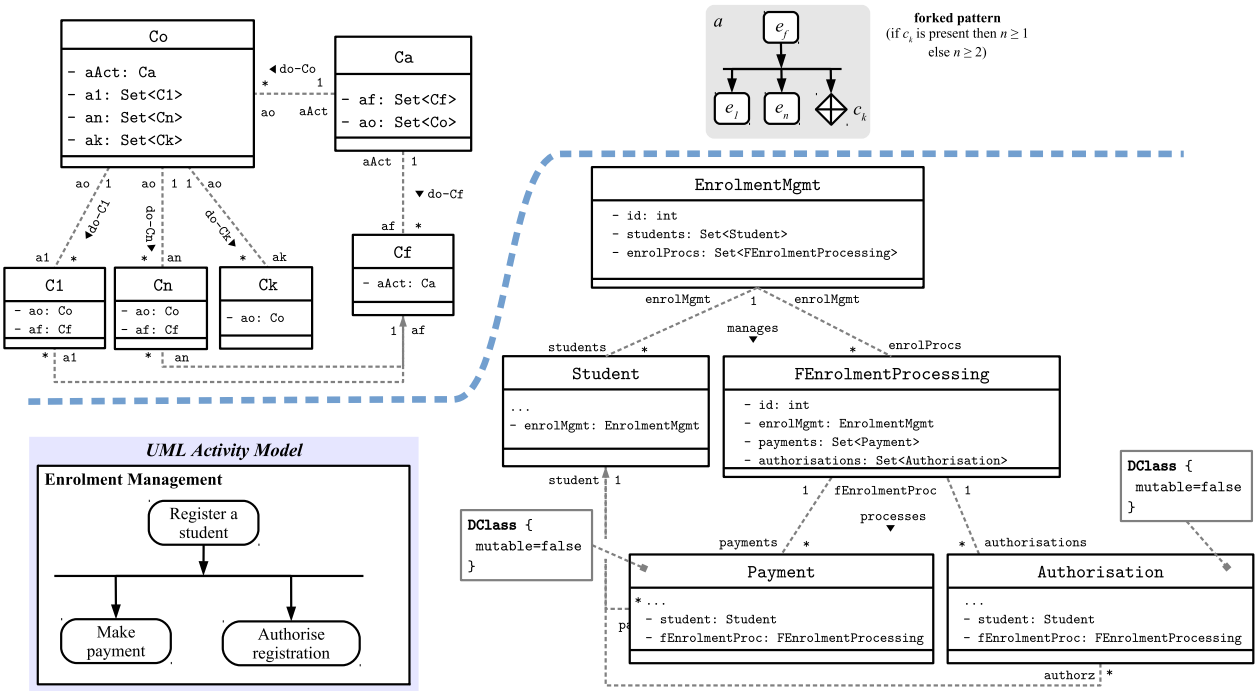

Figure 11: The forked pattern form (top left) and an application to the enrolment management activity.

5.2.3. Forked pattern form

The top-left of Figure 11 is the template model that realises the activity model shown on the top-right. The activity class `Ca` has two associations to the data class `Cf` (referenced by node e_f) and the fork class `Co` (representing the forked node). Class `Co` in turn has associations to the other three domain classes, namely `C1`, `Cn`, and `Ck`. In addition to these, class `Cf` has two weak dependency associations to `C1` and `Cn`, as these classes need to know `Cf` through object passing.

Note a key difference between this template model and the model of the decisional pattern, that class `Ca` is not directly associated to `C1`, `Cn`, and `Ck`. It is instead associated to the control class `Co`, and this class is in turn associated to the other three classes. The rationale behind this design choice is that, compared

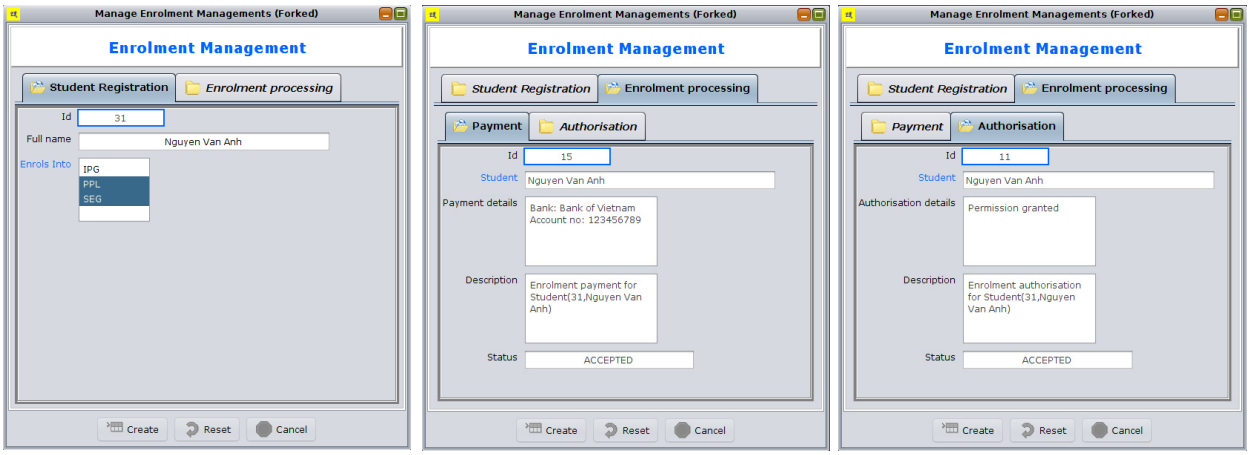

Figure 12: The forked pattern form view of enrolment management activity.

to the decision node, the fork node arguably involves a separate concurrency control process. This process requires a coordination strategy between the action nodes e_1, e_n, and c_k.

Similar to the decisional pattern's model, however, the association from `Co` to `Ck` can be used as a bridge in a larger activity model to connect to other activity flow blocks.

Example

The remainder of Figure 11 shows how the pattern is applied to the enrolment management of CourseMan. The UML activity model of this activity is shown at the bottom left. This activity involves performing student registration (e_f) and then two support actions concurrently: payment processing (e_1) and enrolment authorisation (e_2). These actions must be completed in order for any subsequent actions to proceed.

In this example, `Ca` = `EnrolmentMgmt`, `Cf` = `Student`, `Co` = `FEnrolmentProcessing`, n = 2, `C1` = `Payment`, C2 = `Authorisation`. Note that in addition to the newly added associations in the example model, we have `DClass(Payment).mutable=false` and `DClass(Authorisation).mutable=false`. This is because both `Payment` and `Authorisation` objects are not to be modified after creation. These objects are created by the system after it has executed the payment and authorisation processes.

The three snapshots of the domain model's GUI are shown in Figure 12: one snapshot for one action. The first snapshot is for the first action (student registration). The second and third snapshots are for the two concurrent actions (*resp.*): making payment and enrolment authorisation. A difference between this GUI and the GUIs of the previous two patterns is that it has a 2-level containment. This design is due to a property called model reflectivity, which we will explain in Section 6. Basically, this 2-level containment reflects the length-2 association chain from `EnrolmentMgmt` (the activity class) to `Payment` and `Authorisation`. The first-level containment is that between the activity's GUI and `Student`'s and enrolment processing's GUI. The second-level containment is that between enrolment processing's GUI and `Payment`'s and `Authorisation`'s GUI.

6. Domain-driven software generation

A central part of our L3D method is the use of a reusable software prototype that enables the development team to iteratively and interactively (together with the domain experts) build the domain model. Further, this prototype is reusable during software development to help quickly develop the (final) production software. Our study of the software generator tools developed by the DDD framework providers [5–7] (hereafter referred to as "DDD tools") together with our own experience in developing a software tool named jDomainApp [29] had led us to the identification of four essential properties which we argue a typical domain-driven software needs to have: instance-based GUI, model reflectivity, modularity, and generativity.

In this section, we first introduce a software model that results from our method. We then use this model to define the properties. We demonstrate, through the CourseMan example, how the software generated by jDomainApp supports each property. We conclude the exposition of each property with a brief remark about the limitations of the existing DDD tools in supporting the property. We have been using jDomainApp in both teaching and in developing real-world software projects. A detailed technical description of our L3D implementation in jDomainApp and how to run the tool with CourseMan is available online [31]. To ease discussion, we will use the term "software" to refer to software prototype and the term "user" to refer to the development team members.

6.1. An overview of the software model

The software model of our method is shown in Figure 13. It consists of three layers: domain model layer (core), module layer (middle) and software layer (top). This model realises the DDD's layered architecture [1, 2] with a detailed structure for each layer and the relationships between concepts of the different layers. Given a domain model, we semi-automatically construct the software model and use it as input in jDomainApp to generate a GUI-based software. For example, the software generated for the CourseMan's domain model is shown in Figure 14. In our expositions of the software properties in the subsequent sections, we will give further details about the three layers of the model.

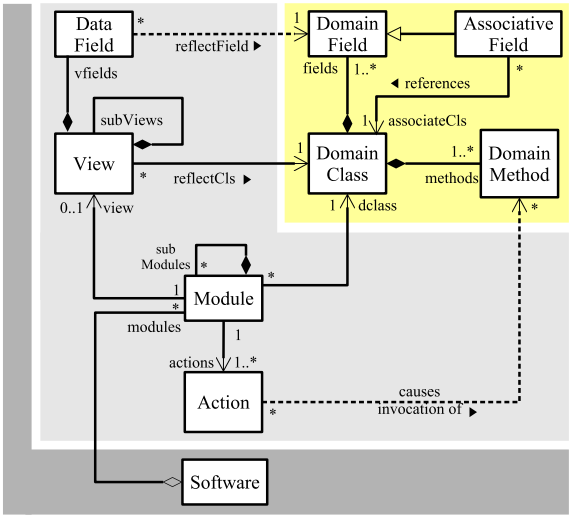

Figure 13: A high-level UML-based software model: (core layer) domain model, (middle layer) module, and (top layer) software.

6.2. Instance-based GUI

This is the extent to which the software uses a GUI to allow a user to observe and work on instances of the domain model. This GUI is necessary in providing an intuitive and a functional presentation of the domain model. This property and the next property (model reflectivity) describe the components at two reflective ends that can be mapped through one structure in the module layer of our software model. We will explain this structure in the next subsection. To illustrate the property, let us consider the GUI of CourseMan's domain model in Figure 14. This GUI consists of a main window that displays a set of menus, a tool bar, and a workspace. The workspace displays the GUI of a set of instances of the domain model. The menus and tool bar provide user with a set of actions that can be performed on this GUI. Again, we will explain the components of this GUI in more detail in the next subsection.

Existing DDD tools mainly focus on providing object GUI and not instance-based GUI. The former focuses on presenting objects of individual domain classes, while the latter targets objects of different domain classes that are associated (directly or indirectly) in the model. The objects and their links constitute a model instance.

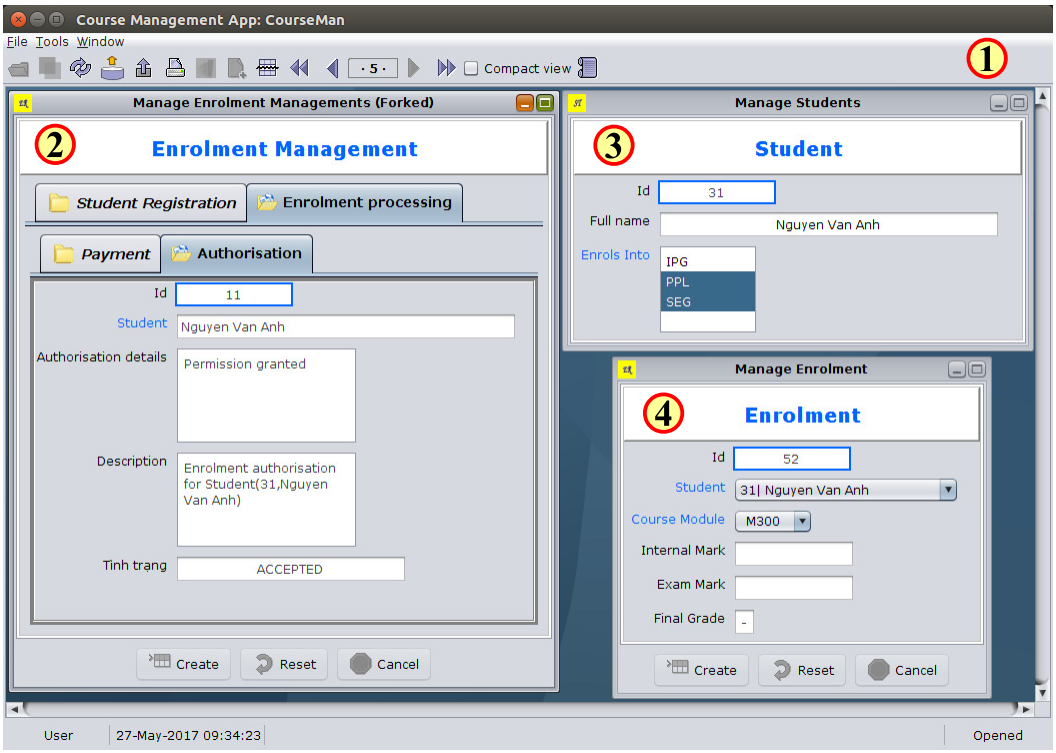

Figure 14: The GUI of CourseMan software prototype generated by jDomainApp: (1) main window, (2-4) the domain model's GUI for `EnrolmentMgmt`, `Student`, and `Enrolment`.

6.3. Model reflectivity

This is the extent to which the GUI faithfully presents the domain model and its structure. This property is central to the usability of the software GUI as a whole. The structure in the top part of the module layer in Figure 13 defines model reflectivity. This part consists of View, Data Field and two associations: reflectField(Data Field, Domain Field) and reflectCls(View, Domain Class).[4] A View is composed of Data Fields, similar to how a Domain Class is composed of Domain Fields. An association chain, which consists of domain classes that are sequentially associated through several associations, is represented by the reflexive compose-of association on View. This association results from two other associations: references(Associative Field, Domain Class) and reflectCls(View, Domain Class). These together cause the presentation of the domain class that is referenced by an associative field of a domain class as a component View (or sub-view) of the View of this class.

For example, the workspace area of Figure 14 contains the view for a (partial) CourseMan's domain model. It consists of the Views of three domain classes: `EnrolmentMgmt` (the activity class discussed in Section 5.2.3), `Student`, and `Enrolment`. Note how the View of `EnrolmentMgmt` closely reflects the structure of the example domain model in Section 5.2.3. First, it consists directly of two sub-views: Student Registration and Enrolment Processing. These two sub-views are created from the two associative fields `EnrolmentMgmt.students` and `EnrolmentMgmt.enrolProcs`. The sub-view Enrolment Processing presents the fork class `FEnrolmentProcessing`. It consists of two other sub-views: `Payment` and `Authorisation`. These are created from the two associative fields (`payments` and `authorisations`) of this class.

The GUIs of existing DDD tools are only partially reflective because they only focus on presenting a GUI for each domain class. Their GUIs at best support the directly associated domain classes. They do not support association chain.

[4]We use normal font for these high-level concepts, because they may not be mapped directly to implementation classes.

6.4. Modularity

In principle, modularity is the extent to which a software development method possesses the following five criteria [19]: decomposability, composability, understandability, continuity, and protection. We adapt these criteria for software model as follow. *Decomposability* is the extent to which the model facilitates the decomposition of a software into smaller, sufficiently-independent modules that are connected by a simple structure. *Composability* is the extent to which the model facilitates construction of modules which may freely be used to produce other software. *Understandability* is the extent to which the model facilitates a human user in understanding a module, without having to know about other modules. *Continuity* is the extent to which the model facilitates making changes to a small number of modules, in response to a small change in the domain requirement. *Protection* is the extent to which the model facilitates localising the effect of abnormal run-time condition that occurs in a module.

We note that these are high-level descriptions which aim to provide a conceptual understanding of the criteria. As such, they contain some quasi-quantitative terms, such as "sufficiently-independent", "simple", "freely", and "small". Although in practical software projects it may be worth establishing exactly what these terms mean, such task is beyond the scope of this paper.

Before discussing these criteria in detail, let us first explain the remaining part of the module layer. This bottom part consists of two concepts: Module and Action. Conceptually, a Module is a self-contained software unit [19] that consists of a domain class, a view that renders objects of this class on a user interface, and a set of actions that are performed on this view. Specifically, a Module has three parts: a Domain Class, a View and a set of Actions. Further, a Module may be composed from one or more other Modules. This composition relationship is mapped to the view-subview composition relationship discussed in the previous subsection. That is, the views of the component modules of a module are sub-views of this module's. Concept Action represents the view actions that are performed on a View and its Data Fields. The execution of an Action may cause the invocation of Domain Methods, each of which belongs to one Domain Class.

For example, the views of the three domain classes in Figure 14 are parts of three modules that are defined from these classes. We implemented these modules in jDomainApp as part of our software prototype for CourseMan. In this, the Module's structure is realised as a tree-based, MVC architecture [26, 36]. Module `EnrolmentMgmt` is composed of two component modules. These are modules of the two sub-views Student Registration and Enrolment Processing. The latter module is in turn composed of two component modules, which are of the two sub-views `Payment` and `Authorisation`.

Now let us define software modularity in terms of our software model. Decomposability is the extent to which the domain classes of the domain model and the modules are constructed in the incremental, top-down fashion (e.g. by adopting the decomposition by abstraction approach [25]). Composability is the extent to which packaging domain classes into modules helps ease the task of combining them to form a new software. Understandability is the extent to which the module structure helps describe what a module is. Since the domain class lies at the core of this structure, to understand a module requires to understand this domain class and how it relates to other parts of the module structure. Continuity is the extent of separation of concerns supported by the module structure. This separation of concerns provides a basis for mapping the requirement to part(s) of each module structure and the interface between the modules. Using this mapping, a designer can identify which and to what extent(s) part(s) need to be updated in response to a change. As for protection, it is the extent by which the domain class behaviour and the user actions concerning the performance of this behaviour are encapsulated in a module. This encapsulation should include a mechanism for handling the abnormal conditions that arise from the module execution.

We note that our software model helps provide a basis for further research into precisely defining the quasi-quantitative terms mentioned at the beginning of this section. For example, because the number of modules is measured based on the domain model's structure and size, decomposibility would be quantified based on these factors.

6.5. Generativity

This refers to the extent to which the software is automatically generated from the domain model, leveraging the capabilities of the target OOPL platform. The higher the generativity is, the more productive the model building process becomes.

In our software model, we define generativity in terms of its three layers as follow. First, **view generativity** is the extent to which the View of a Domain Class is automatically generated. This generativity is enabled by the model reflectivity component of the module layer. Second, **module generativity** is the extent to which a Module is automatically generated from its three components. This generativity is enabled by both the domain model and module layers. Third, **software generativity** is the extent to which a software is automatically generated from its modules. This is enabled by the software layer, which shows how a Software is aggregated from a nonempty set of Modules.

For example, the CourseMan software shown in Figure 14 is generated from three modules (discussed in the previous subsection). Each module is automatically generated from a module configuration, which specifies the configurations for the three module components. Part of this configuration is a view configuration, that is used to automatically generate the view of each module. We discuss the module configuration method in a separate paper [28].

To the best of our knowledge, the existing DDD tools support view and, to some extent, software generativity. They do not support module generativity.

7. Evaluation

In this section, we present our evaluation result of DCSL in the DDD context. We begin with an overview of the evaluation approach.

7.1. Evaluation approach

We evaluate DCSL from two main perspectives: language and generativity support.

7.1.1. Language evaluation

We consider DCSL as a specification language and adapt from [37] the following three criteria for evaluating it: expressiveness, required coding level, and constructibility. Constructibility is evaluated separately from the other two criteria. We discuss how the DCSL's concepts and terms are mapped to the DDD patterns. Further, we compare DCSL to the annotation-based extensions of two DDD frameworks and to the commonly-used third-party annotation sets. To ease notation, we label the annotation-based extensions of the two frameworks as follow: ApacheIsis [6] is labelled **AL**, while OpenXAVA [7] is **XL**.

We use DCSL's terms as the base for evaluation because, as will be explained shortly below, they correspond to the essential terms that are used in the relevant modelling and OOPL literatures and there has been no other annotation-based DSL that supports a similar set of terms. Using the DCSL's terms, we analysed the relevant technical documentations of AL, XL, and DDD patterns to identify the language constructs that are either the same as or equivalent to the primitives or combinations thereof that make up each term. We also made some effort in our analysis to quantify the correspondences.

Expressiveness

This is the extent to which a language is able to express the properties of interest of its domain [37]. We first discuss the *minimality* of DCSL. After that, we evaluate DCSL's expressiveness in three aspects: *structural modelling*, *behavioural modelling*, and *language definition*. For structural modelling, we use the four DCSL's terms as criteria: *domain class*, *domain field*, *associative field*, and *domain method*. For behavioural modelling, we use *activity domain class* as the main criterion. For language definition, we use two core language features as criteria, namely *constraint* and *structural mapping*.

We wish to emphasise that our expressiveness evaluation be interpreted only in terms of the *essential* language features, not in terms of *all* the features. The aforementioned aspects and criteria correspond to the generic and essential terms that are used in the relevant modelling and OOPL literatures. The fact that they are supported in DCSL results from our concious design decision to focus on the essential concepts that are found in the literatures. Specifically, structural and behavioural modelling are two core modelling aspects supported by UML [16]. The structural modelling criteria are primitive domain terms that are derived directly from the four core OOPL's metaconcepts. As will be discussed later in Section 7.2.1, these

are essential terms that are needed to design the domain model of practical software. The activity domain class criterion is key to behavioural modelling using UML activity diagram.

Required Coding Level

This is the extent to which the language allows "...the properties of interest to be expressed without too much hard coding" [37]. Based on the expressiveness evaluation result, we use as criteria three domain terms (domain class, domain field and associative field) that are supported by DCSL, AL and XL. More specifically, we use two sub-criteria for measuring required level of coding: *max-locs* (max number of lines of code) and *typical-locs* (typical number of locs). The lower the values of these the better. Because DCSL, AL and XL are all internal to a base OOPL, we consider annotation property as a loc. Thus, max-locs (*resp.* typical-locs) is the maximum (*resp.* typical) number of properties needed to express a typical domain term. The typical number of properties include only properties that either do not have a default value or are typically assigned to a value different from the default. For example, to specify in DCSL a typical domain field (say `Student.name`) maximally (*resp.* typically) requires these (*resp.* one of these) three `DAttr`'s properties: `length`, `min`, `max`.

Constructibility

This is the extent to which the language provides "...facilities for building complex specifications in a piecewise, incremental way"[37]. For DCSL, we define constructibility as generativity support for domain modelling and discuss this separately in the next section.

7.1.2. Generativity support

The objective of this evaluation is to measure the extent to which DCSL supports domain model generativity. We emphasise that this is different from the software generativity property discussed in Section 6.5. Specifically, we evaluate function BSpaceGen (defined in Section 4.5) in two aspects: behaviour generation and performance. For the first aspect, we use two criteria: *number of domain methods* and *quality*. The latter is measured in terms of the conformance to a *de facto* naming convention of the target OOPL (e.g. JavaBean). To measure these criteria, we use Java as the target host OOPL and, for comparison, a compact CourseMan solution model that we had manually and independently developed for teaching. This solution model consists of three classes (`Student`, `Enrolment` and `CourseModule`), which capture all the essential features of DCSL.

7.2. Expressiveness

We present in this section the evaluation of the expressiveness criterion.

Table 3: (A-left) Comparing DCSL to DDD patterns; (B-right) Comparing DCSL to AL and XL.

Aspects	**DCSL concepts & terms**	**DDD patterns**
Structural modelling	Domain Class	Entity & Aggregate
	Domain Field	
	Associative Field	
	Domain Method	
	Immutable Domain Class	Value Object
Behavioural modelling	Activity Domain Class	Service
Language definition	✓	✘

Aspects	**Expressiveness criteria**	**DCSL**	**AL**	**XL**
Structural modelling	Domain Class	**1/1**	1/1	0/1
	Domain Field	**8/8**	4/8	5/8
	Associative Field	**7/7**	0/7	1/7
	Domain Method	✓	✘	✘
Behavioural modelling	Activity Domain Class	✓	✘	✘
Language definition	Constraint	✓	✘	✘
	Structural mapping	✓	✘	✘

7.2.1. On the minimality of DCSL

We qualitatively argue that DCSL, as defined in this paper, is *minimal* with regards to the set of state space constraints and the essential behaviour that operate on them. First, the state space constraints are identified, based on the authoritative system and software engineering sources [16, 23–25], as being the most fundamental. Second, we have two arguments for why our proposed `OptTypes` constitute a minimum

behaviour specification for domain class. The first argument is that these `OptType`s are a specialisation of three essential behavioural types defined in [24, 25], namely, creator, mutator and observer. Clearly, we need at least these three types of operations in order to create and use objects at run-time. The second argument is that these `OptType`s are needed to accommondate the structural mapping to the aforementioned state space.

7.2.2. Comparing to DDD patterns

The first three rows of Table 3(A) show a mapping between DCSL's concepts and terms and the related DDD patterns discussed in [1, 38]. The DCSL terms form a detailed design language, which realises the high-level design structures described in the DDD patterns. Specifically, the four DCSL's terms are mapped to two DDD patterns (Entity and Aggregate). Concept Immutable Domain Class, which is a special case of Domain Class, is mapped to the Value Object pattern. Concept Activity Domain Class is mapped to the Service pattern.

The last row of the table, however, shows a key difference: while we define DCSL as a design language, the DDD patterns do not constitute a language.

7.2.3. Comparing to DDD frameworks

Table 3(B) presents the evaluation table between DCSL and AL and XL. The fractions in the table are ratios of the number of essential properties of the meta-attribute involved in a DCSL's term/concept that are supported by AL or XL. The denominator of a ratio is the total number of essential properties. A detailed comparison data table is given in Appendix C. For example, the ratio 4/8 for AL *w.r.t* the term Domain Field means that AL only supports 4 out of the total of 8 properties of the meta-attribute `DAttr` (used in Domain Field). The four AL's properties are: `Column.allowsNull`, `Property.editing`, `PrimaryKey.value`, and `Column.length`

Table 3(B) shows that DCSL is more expressive than AL and XL in both structural and behavioural modelling aspects. These two languages only partially support structural modelling and they do not support behavioural modelling using the activity domain class. In particular, AL and XL's support for Associative Field is very limited compared to DCSL.

7.2.4. Comparing to third-party annotation sets

AL and XL support the use of third-party annotation sets, which between them include Java Persistence API (JPA) [39], Java Data Objects (JDO) [40], Hibernate Validator (HV) [41] and Bean Validation (BV) [42]. BV is defined as a standard, of which HV is a reference implementation. In the comparison in Table 3(B), we compared and contrasted DCSL with a subset of the combined annotation set of the above annotation sets that are supported by AL and XL. For completeness, let us compare and contrast DCSL with the full BV's built-in annotation set. We choose BV because it subsumes HV and, similar to DCSL, it is storage-indendendent (JPA and JDO are technologies for mapping domain objects to a data source).

Similar to DCSL, BV's annotations specify constraints on class, field, and method. A number of built-in BV's annotations are similar or comparable to DCSL. For example, `Null` is expressible by `DAttr.optional`; `Min`, `Max`, `Negative`, `NegativeOrZero`, `Positive` and `PositiveOrZero` are expressible by `DAttr.min`, `max`; and `Size` (for String data type) is similar to `DAttr.length`. The `Size` constraint for collection-typed field is subsumed by our DCSL's representation using the property pair `Associate.cardMin`, `cardMax`.

However, BV does not specify equivalent annotations to these properties in DCSL: `DAttr.mutable`, `unique`, `id`, and `auto`. On the other hand, BV specifies the following annotations that are not currently included in DCSL: `Pattern` and `Email` (for string type), `AssertTrue` and `AssertFalse` (for boolean type), `Future` and `Past` (for date type), and `DecimalMin`, `DecimalMax` and `Digits` (for numerical type). Since these annotations describe constraints that concern specific value subsets of some data types, we argue that they form an extension annotation set and be added when the need arises. What we currently focus on in DCSL is the essential annotation set.

Apart from these, we observe three underlying differences between BV and DCSL. First, BV is not defined as a language. As such, it is not clear what the underlying constraints mean in terms of the class

structure. Second, BV does not identify which constraints are essential and which are not. Third, it does not modularise the annotations in terms of the state and behaviour spaces and thus lacks the identification of the structural mapping between the annotations in these two spaces. This also means that, unlike DCSL, BV does not specify a technique for automatically generating the behaviour specification.

7.3. Required coding level

Tables 4(A) and 4(B) respectively show the values of max-locs and typical-locs for the three underlying DCSL's terms that are supported by AL and XL. The last columns of the tables show the total values. The detailed comparison data are presented in Appendix D.

Table 4: (A-left) Summary of max-locs for DCSL, AL and XL; (B-right) Summary of typical-locs for DCSL, AL and XL.

	Max-locs criteria			***Total***
	Domain Class	**Domain Field**	**Associative Field**	
DCSL	1	3	7	***11***
AL	2	4	0	***6***
XL	2	6	1	***9***

	Typical-locs criteria			***Total***
	Domain Class	**Domain Field**	**Associative Field**	
DCSL	1	1	7	***9***
AL	2	1	0	***3***
XL	2	1	1	***4***

It can be observed from both tables that, compared to AL and XL, DCSL has the highest total max-locs (11) and typical-locs (9). However, a closer inspection shows that the DCSL's subtotals for Domain Class and Domain Field (4 and 2 *resp.*) are actually lower than the corresponding subtotals for AL (6 and 3) and XL (8 and 3). Hence, the single contributing factor to DCSL having the two highest totals is the set of 7 mandatory properties needed to express Associative Field. Since all 7 properties are essential for representing this type of field, we conclude that the increase in DCSL's required coding level is a reasonable price to pay for the extra expressiveness that the language enjoys over AL and XL.

7.4. Behaviour generation

Table 5: Behavioural generation statistics for CourseMan.

Domain classes	**Number of methods**	
	Manual	**Generated**
`CourseModule`	20	22
`Enrolment`	17	18
`Student`	19	20

Table 5 presents the evaluation result of BSpaceGen. The two columns "Manual" and "Generated" show the numbers of domain methods in the CourseMan solution and generated models, *resp.*. We exclude AL and XL from this evaluation because they do not support behavioural specification generation at all.

The table shows that BSpaceGen is able to generate all the domain methods in the solution model, plus a few extra methods. Qualitywise, the generated method headers follow the JavaBean convention.

Interestingly, the extra methods generated by BSpaceGen in all three domain classes of CourseMan do not have any domain-specific requirements. Thus, the fact that the manual solution model does not contain these methods does not affect the completeness of the model. For example, both `CourseModule` and `Student` are generated with a `LinkUpdater`-typed method named `onUpdateEnrolments`. However, these methods do not have any domain-specific logic defined for them, because neither `CourseModule` nor `Student` object state needs to be updated when a linked `Enrolment` is changed.

The generativity support of DCSL is amplified by the fact that BSpaceGen is also applicable to generating the behavioural specification of the activity domain class.

7.5. Performance analysis

Since AL and XL do not support code generation, we focus our performance analysis on function BSpaceGen. Based on the linear complexity result of Alg. 1 (presented in Section 4.5), we conclude that BSpaceGen is practically capable of handling domain classes with large state spaces.

7.6. Discussions

Let us conclude our evaluation with a number of remarks about the design and implementation of DCSL and its use in software design. The first remark is about the *granularity of domain method*. In DCSL, we focus on the primitive domain methods which are structurally mapped to a set of essential state space constraints. The question then arises as to how to model more coarse-grain methods, such as those discussed in [1, 2] that realise the domain behaviours. The behavioural modelling patterns discussed in Section 5 are a first important step towards addressing this question. However, more research should be carried out in this direction.

The second remark concerns the *abstraction level*. In essence, DCSL uses the syntax structure of the base OOPL to express the essential domain terms. It could be argued that a more preferred method of abstraction is to use the domain terms directly as metaconcepts of the language. However, this requires a better support from the base OOPL, which is currently lacking in both Java and C#.

The third remark concerns the *sublanguage* L_O of the base OOPL, from which DCSL is extended. The DCSL's annotation-based extension does not make any changes to the existing attributes of the metaconcepts Class, Field, Method, and Annotation. The OCL constraints of DCSL, however, do require introducing into these metaconcepts a number helper features (presented in the extended paper [21]). These helper features can be implemented either as such or as part of the tool kit class named `Tk`. We show in [21] how this class is used to define some (other) helper features.

The fourth remark is about the *minimality*, *behaviour-essentiality* and *evolution* of DCSL. We judged minimality based on a set of fundamental constraint requirements and essential behaviours that are taken from a set of authoritative academic and practical resources. However, we would not claim that these resources are universal to every researcher and practitioner in the field. It is thus our hope that our proposed 'minimality' attribute for an annotation-based DSL for domain class specification would attract interests and invite feedbacks from those concerned, with an aim to come to a shared and improved understanding.

Regarding to the *behaviour-essentiality* of DCSL. We basically took the generic "behaviourally-complete" property proposed in [5] for domain models in general, extracted a set of essential underlying features (called "behaviour essential") and then used this as the starting point for designing DCSL. This way, DCSL can be used to define many different behaviourally complete domain models.

As with any language, DCSL starts out as a small language with just a set of essential features. We expect that DCSL, as it is put into use, will evolve to incorporate more features. Some of these features perhaps will surface as we investigate how to address nonfunctional requirements, some of which are mentioned in the future work in Section 9.

The fifth remark is about the *modularity* of DCSL design. According to Hudak [43], a modular DSL is one that has a modular language specification structure, which expresses the domain requirements using layers of abstraction. In DCSL, we first focus on the lowest level of abstraction and specify the primitive constraints needed to express the essential state and behaviour spaces of domain class. We then look at how to put the primitive modelling elements together to form larger structures. These structures are defined by the structural mapping rules between these two spaces.

The final remark is about *domain model evolution* when the production software evolves. This happens when the software is put into use and needs to undergo maintenance at some point. Given that the software is developed using our proposed software model, software evolution can basically be handled as part of our overall L3D method as follow (a conceptual architecture of this idea is presented in [28]). First, we identify the necessary software changes through interaction via the software GUI. Then, we map out these changes (via the module layer structure of the software model) to the corresponding changes that need to be applied to the domain model. However, more research need to be conducted concerning how to automate this procedure using the software tool.

8. Related work

We position our work in the intersections between the following areas: DSL engineering, DDD, MDSE, attribute-oriented programming (AtOP), behavioural interface specification language (BISL), and aspect oriented software design (AOD).

DSL engineering

DSL [44–46] is a software language that is specifically designed for expressing the requirements of a problem domain, using the conceptual notation suitable for the domain. An important benefit of DSL is that it raises the level of abstraction of the software model to the level suitable for the domain experts and thus helps them participate more actively and productively into the development process.

DSLs can be classified based on domain [11] or on the relationship with a host language [10, 44, 46] (e.g. OOPL). From the domain's perspective, DSLs are classified as being either vertical or horizontal. A vertical DSL, *a.k.a* business-oriented DSL, targets a bounded real-world domain. In contrast, a horizontal DSL (*a.k.a* technical DSL) targets a more technical domain, whose concepts describe the patterns that often underlie a class of vertical domains which share common features. From the host language's perspective, DSLs are classified as being internal or external (*a.k.a* compiled [45]). In principle, internal DSL has a closer relationship with the host language than external DSL. A typical internal DSL is developed using either the syntax or the language tools of the host language. In contrast, a typical external DSL has its owns syntax and thus requires a separate compiler to process.

Our proposed DCSL is a type of fragmentary, internal and horizontal DSL. The shared features that are captured in DCSL are those that form the domain state space. In fact, the term fragmentary, internal DSL in [10] is analogous to another term *annotation-based language*, which is proposed in a recent work [9]. The latter defines three types of annotation-based language, one of which (namely annotation-based language extension) fits the characteristics of DCSL. However, unlike our work [9] does not discuss how metamodelling (such as using UML and OCL) is used to define the abstract and concrete syntax of a language. Further, the illustrative example for annotation-based language extension is a simple language which consists of just one annotation (Override). Our DCSL contains an essential set of annotations that are specifically used for domain class specification and for aiding software generation.

DDD

Both the foundational work [5] and the consequential DDD method [1] describe the basic principles for designing object-oriented software. Our L3D method extends DDD with the use of DCSL for constructing the domain model of software. Our method precisely defines DCSL's ASM using OCL and extends two previous work [17, 18], which present some preliminary results on the design and use of DCSL in DDD. What is distinctive about DCSL, compared to [5], is the idea of behaviourally-essential domain class.

The idea of combining DDD and DSL to raise the level of abstraction of the target code model has been advocated in [10] by both the DDD's author and others. However, [10] does not discuss any specific solutions for the idea. Other work on combining DDD and DSL (e.g. Sculptor [47]) focus only on structural modelling and use an external rather than an internal DSL. We argue that internal DSL is essential for the feasibility of the domain model. In addition, our method addresses both structural and behavioural modelling.

DDD frameworks. There are two DDD frameworks (namely ApacheIsis [5, 6] and OpenXava [7]) that adopt the DDD approach. As discussed in this paper, these frameworks are similar to our approach in the use of an annotation-based extension to construct the domain class model. However, they do not define the extension into a language. More importantly, their extensions differ from DCSL in the expressiveness of the annotations and the required coding level. In addition, their extensions, though include many annotations, do not identify those that express the essential state space constraints like ours.

Another main difference is that our L3D method uses DCSL to more completely support behavioural modelling. The DDD frameworks only partially support behavioural modelling in the form of method-based action in domain class. They lack support for the activity class and the activity flow through the actions.

With regards to tool support, our work is similar to the DDD frameworks in the use of a software generator (*a.k.a* smart code generator in [8]) that automatically generates a software prototype from the

domain model. The prototype uses some form of user interface, that is mapped to a domain class in the model. We argue that our tool is relatively "smarter" in that it is based on a 4-property characterisation that helps determine the extent to which the generated software can effectively be used by the development team to build the domain model. The software generated by the DDD frameworks have several limitations when scrutinised under the four properties that we propose.

MDSE

Similar to DDD, our L3D method shares with MDSE [8] the idea of using models to design software. However DDD differs from MDSE in two main aspects. First, DDD addresses a more specific problem of domain modelling for software development, while MDSE's scope is wider and covers system engineering in general. In particular, DDD advocates the use of a layered architecture style, at the core of which lies the domain model. Second, while DDD acknowledges the role of model engineering, it also highlights the importance of a ubiquitous language that is used by all the human actors in the process.

Our L3D method is conceived for DDD, but can also be used as part of an MDSE method. As far as DDD is concerned, our method uses DCSL to raise the level of the abstraction of the base OOPL to make it fit better for use with the ubiquitous language. By supporting two popular high-level base OOPLs, one of which (Java) is a platform-neutral language, DCSL is applicable to DDD for a wide variety of practical software domains. As far as MDSE is concerned, DCSL can be used to construct OOPL-specific models. Further, DCSL's mapping to a UML-based (external) DSL can be established (e.g. see [17]). This mapping is used to engineer transformation from the platform independent models.

The idea of combining MDSE with DSLs is formulated in [8, 11]. This involves applying the metamodelling process to create metamodels of software modelling languages (include both general-purpose languages and DSLs). The work in [11] proposes two variants of the metamodelling process: for vertical DSLs, it is the domain modelling process (discussed in [10]); for horizontal DSLs, it is a pattern-based modelling process. Our work in this paper adopts the pattern-based metamodelling approach, but targets internal DSL.

DSLs for software modelling. The method in [48, 49] proposes the use of a combination of DSLs to build a complete software model. In these works, a 3-step development method is outlined, which includes: (1) determine the application architecture, (2) develop the DSLs that fit this architecture and (3) combine the DSLs by defining transformations between them. A key feature of this method is that the DSLs are designed to address different parts of the architecture, and that each DSL is used to create not one monolithic model but multiple partial models. The adopted architecture is a layered, service-oriented architecture named SMART-Microsoft. Four DSLs are defined for this architecture: web scenario DSL (for the presentation layer), data contract DSL (data contract layer), and service and business class DSLs (business layer).

Our demonstration of the L3D method using the software generation tool [28, 29] shows how our method is ultimately similar in spirit to the above work. Within the scope of this paper, however, we would reveal two main differences between DCSL and the business class DSL. First, DCSL is an internal rather than an external one. Second, DCSL seeks to define a minimal state space and the essential behaviour that operates on this space. The business class DSL does not address this.

Structural mapping and behaviour generation. Our proposed structural mapping is similar in spirit to [50], which is studied in the context of UML class diagram. The base operations that are generated in this work are identified based on structural events. A structural event is a basic change to the system state. The basic operations that are identified for each class in a class diagram include (1) object creator, (2) object deleter, (3) attribute value updater, (4) link adder, (5) link deleter, (6) object generaliser, and (7) object specialiser. An operation is declaratively defined with an operation contract that includes a post-condition clause. This clause is defined with consideration to any dependencies that the operation's structural event has on other events.

The set of base methods that are generated through our structural mapping and the basic operations of [50] are mappable to the set of CRUD operations. According to Brambilla et al. [8], these operations are commonly used in real-world software. Our base methods overlap with [50] in terms of these four operations: (1), (3), (4) and (5). We consider operations (2), (6), and (7) as being the responsibilities of a system class (e.g. object manager [29]). Further, both our work and [50] provide OCL-based definitions for the pre- and post-conditions of the generated behaviour.

The key differences between our structural mapping and [50] are the following. First, our mapping rules are established directly between the two design spaces of the structural schema. Second, we do not use structural events but the state space constraints to determine the mapping. Third, our constraints include some generic constraints (incl. `mutability` and `optionality`) and some domain-specific constraints (`id` and `auto`) that are not supported in [50]. We showed how these constraints are essential to the identification and definition of some essential operations not found in [50]. Fourth, the domain method behaviour induced by our structural mapping can be written more concisely using annotations, as opposed to being written in OCL constraints in [50].

AtOP

Our idea of using annotation to represent state space constraints is inspired by attribute-oriented programming (AtOP) [13–15, 51]. As discussed above, the existing DDD frameworks also adopt this representation. In principle, AtOP extends a conventional program with a set of attributes, which capture application- or domain-specific semantics [13], [51]. These attributes are represented in contemporary OOPLs as annotations. It is shown in [14], with empirical evidence, that because programmers' mental models do overlap the practice of recording the programmer's intention of the program elements using annotations is rather natural. Further, the use of purposeful annotations in programs help not only increase program correctness and readability [14] but raise its level of abstraction [15].

Using AtOP for DDD and MDSE. The relationship between AtOP and DDD is manifested in the use of annotations in DDD. The two DDD frameworks discussed earlier clearly demonstrate the benefits of such use.

With regards to the use of AtOP in MDSE, a classic model of this combination is used in the development of a model-driven development framework, called mTurnpike [12]. This framework combines AtOP with model-driven development in a top-down fashion, with an aim to define domain-specific concepts at both the modelling and programming levels. At the modelling level, the concepts are expressed in a UML-based domain specific model (DSM). At the programming level, the concepts are represented as a domain-specific code model (DSC), which are attribute-oriented programs. DSM and DSC are transformed automatically to one another via a set of pre-defined, semantic-preserving rules. The framework also supports hand-written code, which is written directly by programmer in bodies of the methods found in the DSC. This code is combined with the platform-specific models of the DSM and DSC to generate the final code.

Since this classic model is a special case of MDSE, our earlier comparison between L3D method and MDSE also applies to [12]. It is worth emphasising further that we focus on the development of annotation-based DSLs. The models of these DSLs are comparable to the DSCs.

More recently, the work in [15] proposes a bottom-up MDSE approach, which entails a formalism and a general method for defining annotation-based embedded models. More specifically, a class of embedded models *w.r.t* a given model class M (e.g. class diagram) and a given OOPL G (e.g. Java) is the tuple $\langle A_M, P_G \rangle$, where A_M is the abstract syntax of M and P_G is the code pattern of M in G. This assumes a mapping exists between the metamodels of A_M and G.

Compared to [15], our definition of internal DSL also takes into account the mapping between the base OOPL (G) and a high-level modelling language (A_M = UML class diagram). However, our formalism differs in the notion of a domain-specific OOPL extension. This helps clearly separate the metamodel of the base OOPL from that of an internal DSL.

Beside the above, our method differs from both works [12, 15] in two important ways: (1) the design of a single DSL (DCSL) that can be used to express the essential aspects of both structural and behavioural modelling in a domain model, and (2) how this domain model is used as the core component to automatically generate the entire software.

BISL

Our design of DCSL effectively places this language in the same category as BISL [52–56], which is the class of languages for specifying software behaviour. At the unit level, one specifies the behaviour for either a method or a type (class or interface). A behaviour specification includes two parts: interface definition and behaviour description. The former consists in the information needed by other software units. The

latter specifies one or more state transformations that result from executing the behaviour.

BISLs can be characterised by the properties that they can express and by the software development artefacts that they describe [52]. Of particular relevance to DCSL are BISLs that express functional behaviour properties for object oriented source code. These properties are typically concerned with the transformation from the pre-condition state to the post-condition state of a method and consistency criteria (invariant) of class. More specifically, DCSL is comparable to JML [53] and Spec# [54]. JML is a BISL for Java and Spec# is one for C#. While DCSL is designed as a DSL with a specific aim in mind, both JML and Spec# are general-purpose languages. The meta-attributes of DCSL are mapped to a domain-specific, sub-set of features supported by both languages, namely pre- and post-condition of method and class invariant. While JML and Spec# provide their own expressions for use in the pre- and post-condition specifications, DCSL uses OCL expression for these. Being a DSL has its advantage in that DCSL's primitives are designed to maximise the generative capability in software development.

Sharing some high-level design terminology with, though not directly related to, DCSL is a formal object-oriented specification language called Object-Z [55]. Object-Z is derived from Z [56], which, according to [52], is an analysis-level specification language. This differs from DCSL, which is a code-level specification language. Another key difference is that DCSL is a DSL, while Object-Z is a general purpose language. Nontheless, our two design terms, namely domain state space and behaviour space, are structurally similar to state and operation schemas of Object-Z (*resp.*). State schema is a structure for defining the state variables of a class and the constraints on those variables. Together, these define the valid states that an object may be in at any given point in time. On the other hand, operation schema is a structure for defining operations. It specifies the allowable changes to the object state.

AOD

According to Kiczales [57], an aspect is a software system property that can not be represented entirely by a functional unit. Examples of aspect include logging, security, performance, memory access patterns, and so on. Aspect-oriented software design (AOD) is part of a larger theme of aspect-oriented analysis and design. According to Chitchyan et al. [58], AOD has the same aim as any other software design activity; that is "to characterise and specify the behaviour and structure of the software system". A central problem in AOD is how to model the cross-cutting concerns. AOD provides a language and a process for modelling these concerns. A key benefit of AOD is that it helps increase modularity by enhancing module cohesiveness and reducing module coupling.

Conceptually, the cross-cutting concerns would be considered as forming specific domains, for which DSLs are developed. Compared to our method, however, the state-of-the-art AOD approaches reported in [58] do not address the combination of structural and behavioural modelling concerns. In particular, three related approaches (namely AOSDUC, CAM/DAOP, and Activity Diagrams) only address behavioural design using UML activity diagram.

9. Conclusion

In this paper, we have proposed an L3D method for developing object oriented software. The core component of our method is an annotation-based DSL named DCSL, that is used to construct the domain model directly in a base OOPL. We showed how DCSL expresses a set of essential structural (state space) constraints and the essential domain class behaviour. DCSL helps establish a structural mapping between the state and behaviour spaces of domain class. We used this mapping to define a technique for automatically generating the behavioural specification. We also discussed how DCSL enables a technique to express the core behavioural modelling patterns of UML activity diagram. This provides basis for a unified domain model that can then be used as input to automatically generate the software. Another contribution of our work is the proposal for a 4-property characterisation of a typical software that is generated from the domain model. This software is used as a prototype during domain modelling and is reused after this has been completed to develop the production software. We demonstrated the software properties using a Java software tool that we developed for our method. We also evaluated DCSL in the context of DDD.

We thus argue that our method significantly extends the state-of-the-art in DDD with respect to using annotation-based DSL to not only bridge the gaps between domain model and code but enhance the overall quality and generativity of the software that is generated from this model. Our plan for future work includes:

- extending the behavioural modelling patterns further to support the activity graph logic.
- designing an effective mechanism for tackling domain model evolution.
- developing a plug-in in an IDE (e.g. Eclipse) for the method.
- improving the GUI design of the software (e.g. to support more sophisticated GUI layouts).
- developing internal DSLs for other technical domains, including software configuration, security, *etc.*
- quantitatively evaluate the software generated by JDOMAINAPP against the four proposed properties.

Acknowledgements

This work was funded by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2015.25. We would like to thank all the past and present students and teaching staff of the Software Engineering and Special Subject 2 course modules at Hanoi University, who have studied, used and provided useful feedbacks for DCSL. We wish to thank the anonymous reviewers for numerous insightful feedbacks on the first version of this paper.

References

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.
[2] V. Vernon, Implementing Domain-Driven Design, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2013.
[3] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, 4th Edition, Vol. 2, Addison-wesley Reading, 2005.
[4] J. Gosling, B. Joy, G. L. S. Jr, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2014.
[5] R. Pawson, R. Matthews, Naked Objects, in: OOPSLA'02, OOPSLA'02, ACM, New York, NY, USA, 2002, pp. 36–37.
[6] Dan Haywood, Apache Isis - Developing Domain-driven Java Apps, Methods & Tools: Practical knowledge source for software development professionals 21 (2) (2013) 40–59.
[7] J. Paniza, Learn OpenXava by Example, CreateSpace, Paramount, CA, 2011.
[8] M. Brambilla, J. Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice, 1st Edition, Morgan & Claypool Publishers, 2012.
[9] M. Nosál', M. Sulír, J. Juhár, Language Composition Using Source Code Annotations, Computer Science and Information Systems 13 (3) (2016) 707–729.
[10] M. Fowler, T. White, Domain-Specific Languages, Addison-Wesley Professional, 2010.
[11] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st Edition, Addison-Wesley Professional, Upper Saddle River, NJ, 2008.
[12] H. Wada, J. Suzuki, Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, no. 3713 in LNCS, Springer, 2005, pp. 584–600.
[13] V. Cepa, S. Kloppenburg, Representing Explicit Attributes in UML, in: 7th Int. Workshop on Aspect-Oriented Modeling (AOM), 2005.
[14] M. Sulír, M. Nosáľ, J. Porubän, Recording Concerns in Source Code Using Annotations, Computer Languages, Systems & Structures 46 (2016) 44–65.
[15] M. Balz, Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-Based Software Development, Ph.D. thesis, Universität Duisburg-Essen (Jan. 2012).
[16] OMG, Unified Modeling Language version 2.5, Tech. rep. (2015).
URL http://www.omg.org/spec/UML/2.5/
[17] D. M. Le, D.-H. Dang, V.-H. Nguyen, Domain-Driven Design Using Meta-Attributes: A DSL-Based Approach, in: Proc. 8th Int. Conf. Knowledge and Systems Engineering (KSE), IEEE, 2016.
[18] D. M. Le, D.-H. Dang, V.-H. Nguyen, Domain-Driven Design Patterns: A Metadata-Based Approach, in: Proc. 12th Int. Conf. Computing and Communication Technologies (RIVF), IEEE, 2016.
[19] B. Meyer, Object-Oriented Software Construction, 2nd Edition, ISE Inc., Santa Barbara (California), 1997.
[20] A. Hejlsberg, M. Torgersen, S. Wiltamuth, P. Golde, The C# Programming Language, 4th Edition, Addison Wesley, Upper Saddle River, NJ, 2010.
[21] D. M. Le, D.-H. Dang, V.-H. Nguyen, On Domain Driven Design Using Annotation-Based Domain Specific Language [Extended], Tech. rep., VNU University of Engineering and Technology (2017).
URL https://drive.google.com/open?id=1536zs321O66sR2azLgTU_pifjJY7sK-2

[22] D. Akehurst, G. Howells, K. McDonald-Maier, Implementing Associations: UML 2.0 to Java 5, Softw Syst Model 6 (1) (2007) 3–35.
[23] J. A. Hoffer, J. George, J. A. Valacich, Modern Systems Analysis and Design, 7th Edition, Prentice Hall, Boston, 2013.
[24] B. Liskov, J. Guttag, Abstraction and Specification in Program Development, MIT Press, 1986.
[25] B. Liskov, J. Guttag, Program Development in Java: Abstraction, Specification, and Object-Oriented Design, Pearson Education, 2000.
[26] 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), IEEE, 2015, pp. 19–24.
[27] D. M. Le, A Domain-Oriented, Java Specification Language, in: Proc. 7th Int. Conf. Knowledge and Systems Engineering (KSE), IEEE, 2015, pp. 25–30.
[28] D. M. Le, D. H. Dang, V. H. Nguyen, Generative Software Module Development: A Domain-Driven Design Perspective, in: Proc. 9th Int. Conf. on Knowledge and Systems Engineering (KSE), 2017, pp. 77–82.
[29] D. M. Le, jDomainApp version 5.0: A Java Domain-Driven Software Development Framework, Tech. rep., Hanoi University (2017).
URL https://drive.google.com/open?id=0B3C14WpmosISM2MyWUFMRWhzNFE
[30] A. Fuggetta, E. Di Nitto, Software Process, in: Proceedings of the on Future of Software Engineering, FOSE 2014, ACM, New York, NY, USA, 2014, pp. 1–12.
[31] D. M. Le, DCSL implementation in jDomainApp, original-date: 2017-05-28 (May 2017).
URL https://github.com/vnu-dse/dcsl
[32] H.-E. Eriksson, M. Penker, Business Modeling With UML: Business Patterns at Work, 1st Edition, John Wiley & Sons, Inc., New York, NY, USA, 1998.
[33] M. Dumas, A. H. M. t. Hofstede, UML Activity Diagrams as a Workflow Specification Language, in: M. Gogolla, C. Kobryn (Eds.), UML 2001, LNCS, Springer, 2001, pp. 76–90.
[34] D. Riehle, H. Züllighoven, Understanding and Using Patterns in Software Development, Theory Pract. Obj. Syst. 2 (1) (1996) 3–13.
[35] E. Gamma, R. Helm, R. Johnson, J. Vlissides, G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley Professional, Reading, Mass, 1994.
[36] G. E. Krasner, S. T. Pope, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, Journal of object-oriented programming 1 (3) (1988) 26–49.
[37] A. v. Lamsweerde, Formal Specification: A Roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 147–159.
[38] E. Evans, Domain-Driven Design Reference: Definitions and Pattern Summaries, Dog Ear Publishing, LLC, 2014.
[39] Oracle, Java Persistence API Specification (2018).
URL https://jcp.org/aboutJava/communityprocess/final/jsr317/index.html
[40] DataNucleus, JDO Annotations Reference (v5.1) (2018).
URL http://www.datanucleus.org/products/datanucleus/jdo/annotations.html
[41] Red Hat, Hibernate Validator (2017).
URL http://hibernate.org/validator/
[42] Red Hat, Bean Validation 2.0 (JSR 380) (2017).
URL http://beanvalidation.org/2.0/
[43] P. Hudak, Modular Domain Specific Languages and Tools, in: Proc. 5th Int. Conf. on Software Reuse, 1998, pp. 134–142.
[44] A. van Deursen, P. Klint, J. Visser, Domain-specific Languages: An Annotated Bibliography, SIGPLAN Not. 35 (6) (2000) 26–36.
[45] E. Visser, WebDSL: A Case Study in Domain-Specific Language Engineering, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, no. 5235 in LNCS, Springer, 2008, pp. 291–373.
[46] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[47] Sculptor Team, Sculptor - Generating Java code from DDD-inspired textual DSL (2016).
URL http://sculptorgenerator.org/
[48] J. Warmer, A Model Driven Software Factory Using Domain Specific Languages, in: Model Driven Architecture-Foundations and Applications, Springer, Berlin, Heidelberg, 2007, pp. 194–203.
[49] J. Warmer, A. Kleppe, Building a Flexible Software Factory Using Partial Domain Specific Models, in: 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), University of Jyvaskyla, 2006.
[50] J. Cabot, C. Gómez, Deriving Operation Contracts from UML Class Diagrams, in: G. Engels, B. Opdyke, D. C. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems, LNCS, Springer, 2007, pp. 196–210.
[51] V. Cepa, Attribute Enabled Software Development, VDM Verlag, Saarbrücken, Germany, Germany, 2007.
[52] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, M. Parkinson, Behavioral Interface Specification Languages, ACM Comput. Surv. 44 (3) (2012) 16:1–16:58.
[53] G. T. Leavens, A. L. Baker, C. Ruby, Preliminary Design of JML: A Behavioral Interface Specification Language for Java, SIGSOFT Softw. Eng. Notes 31 (3) (2006) 1–38.
[54] M. Barnett, K. R. M. Leino, W. Schulte, The Spec# Programming System: An Overview, in: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2004, pp. 49–69.
[55] G. Smith, The Object-Z Specification Language, Vol. 1 of Advances in Formal Methods, Springer US, Boston, MA, 2000.
[56] J. M. Spivey, An Introduction to Z and Formal Specifications, Software Engineering Journal 4 (1) (1989) 40–50.

[57] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: ECOOP'97-Object-Oriented Programming, LNCS, Springer, 1997, pp. 220–242.
[58] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, A. Jackson, Survey of Aspect-Oriented Analysis and Design Approaches, Report, University of Leicester (May 2015).
URL https://lra.le.ac.uk/handle/2381/32112

Appendix A. The OCL Constraints of DCSL's ASM

Table A.6: The OCL constraints of the DCSL's ASM.

Names	Constraints

WF1	`-- Parameter.ref.value (if specified) refers to name of a Field in same class`
	`context p : Parameter inv :`
	`  not(p.ref.oclIsUndefined()) implies`
	`  p.owner.owner.fields->exists(f | f.name = p.ref.value)`

WF2	`-- Method.ref.value (if specified) refers to name of a Field in same class`
	`context m : Method inv :`
	`  not(m.ref.oclIsUndefined()) implies`
	`  m.owner.fields->exists(f | f.name = m.ref.name)`

WF3	`-- Method.atr is only specified for overridden methods that also have Method.ref`
	`context m : Method inv :`
	`  not(m.atr.oclIsUndefined()) implies`
	`  m.isOverriden() and not(m.ref.oclIsUndefined())`

WF4	`-- all overridden methods of a subtype that reference a domain field must`
	`-- preserve the DAttr of that field`
	`context c : Class inv:`
	`  not(Tk::ancestors(c).oclIsUndefined()) implies`
	`  c.overridenMethods()->forAll(p | not(p.atr.oclIsUndefined()) implies`
	`  Tk::ancestors(c)->exists(c1 |`
	`  c1.fields->exists(a | p.isAttrRef(a) and p.atr.preserves(a.atr)))`
	`  )`

OM	`-- c has a mutable field a`
	`-- exists a mutable field a of c`
	`context c: Class inv Mutable:`
	`  c.dcl.mutable implies c.fields->exists(a | a.mutable)`

FM	`-- exists a method p of a.owner s.t p's result expression is`
	`--  a VarAssignExpression for a`
	`context a: Field inv Mutable :`
	`  a.atr.mutable implies a.owner.methods->exists(p | p.isMutatorRef(a) and`
	`  let e : OclExpression = p.resultExp in`
	`  not(e.oclIsTypeOf(OclVoid)) and e.oclIsTypeOf(VarAssignExpression) and`
	`  e.oclAsType(VarAssignExpression).lhs.referredVariable = a)`

FO

```
-- exists a constructor u of a.owner s.t. none of the body expressions of u
--  contains a in the LHS
context a: Field inv Optional :
  a.atr.optional implies a.owner.methods->exists(u | u.isCreator() and
    u.postExps()->forAll(e | e.oclIsTypeOf(VarAssignExpression) implies
    e.oclAsType(VarAssignExpression).lhs.referredVariable <> a ))
```

FL

```
 -- for all operation p of a.owner that mutates a via a parameter m,
 -- exists an expression e in p's pre that restricts (m.size() <= a.atr.length)
 context a: Field inv MaxLength :
  let l : Integer = a.atr.length in
  l >= 1 implies
  if a.type.name = 'String' then
      a.owner.methods->forAll(p | p.isCreatorOrMutatorRef(a) implies
      p.params->exists(m | m.isAttrRef(a) and
        p.preExps().exists(e | e.equals(m.name+'.size() <= ' + l)) ))
  else
    true
  endif
```

FU

```
-- a has unique values among all objects of the class
context a: Field inv Unique :
  a.atr.unique implies
    a.owner.allInstances()->forAll(o, o' | o <> o' implies
                                  a.value(o) <> a.value(o'))
```

IF

```
-- id means both not(Optional) and Unique
context a: Field inv Id :
  a.atr.id implies not(a.Optional) and a.Unique
```

YV

```
-- for all operation p of a.owner that mutates a via a parameter m,
--  exists an expression e in p's pre that restricts m >= y
context a: Field inv MinValue :
  let y : Double = a.atr.min in
  y <> Double::INFINITY implies
  if a.isNumericType() then
    a.owner.methods->forAll(p | p.isCreatorOrMutatorRef(a) implies
      p.params()->exists(m | m.isAttrRef(a) and
        p.preExps().exists(e | e.equals(m.name + ' >= ' + y)) ))
  else
    true
  endif
```

XV

```
-- for all operation p of a.owner that mutates a via a parameter m,
--  exists an expression e in p's pre that restricts m <= a.atr.max
context a: Field inv MaxValue :
  let x : Double = a.atr.max in
  x <> Double::INFINITY implies
  if a.isNumericType() then
    a.owner.methods->forAll(p | p.isCreatorOrMutatorRef(a) implies
      p.params()->exists(m | m.isAttrRef(a) and
        p.preExps().exists(e | e.equals(m.name + ' <= ' + x)) ))
  else
    true
  endif
```

TF

```
-- a is not mutable and is not optional and exists a constructor u and
--  a private method p (both of a.owner) s.t u contains a body
--  expression e of the form 'a = p(...)'
--  (i.e. u invokes p and assigns the result obtained to a)
context a: Field inv Auto :
  a.atr.auto implies
    let c : Class = a.owner in  -- c is the owner class of a
    if not(a.Mutable) then      -- not Mutable(a)
      if not(a.Optional) then   -- not Optional(a)
      -- there exists constructor u and a private operation p of c and a
      -- body expression e of u s.t. e is a OperationCallExpr of the form
      -- a = p(...)
      c.methods->exists(u, p |
        u.isCreator() and p.visibility = VisibilityKind::private and
         u.postExps()->exists(e1 | e1.oclIsTypeOf(VarAssignExpression) and
         let e = e1.oclAsType(VarAssignExpression) in
            e.rhs.oclIsTypeOf(OperationCallExp) and
            e.rhs.oclAsType(OperationCallExp).referredOperation = p and
         e.lhs.referredVariable.asProperty() = a ))
      else
        false
      endif
    else
      false
    endif
```

YL

```
-- for-all link-remover operation p of c = owner(a) that removes a collection
--  of objects of c2 = a.type, the pre- and post-conditions of p
--  must be set accordingly
context a: Field inv CardMin :
  let y : Integer = a.asc.associate.cardMin in
  y > -1 implies
  let ca : Class = elementType(a.type) in
  a.owner.linkRemovers()->forAll(p | p.isAttrRef(a) implies
    p.params()->size() = 1 and let m : Parameter = p.params().any(true)
    in elementType(m.type) = ca and
      p.preExps().exists(e |
        e.equals(ExprTk::genLinkRemoverPreCond(ca, m, y))) and
      p.postExps().exists(e1, e2 |
        e1.equals(ExprTk::genLinkRemoverPostCond1(a, m)) and
      e2.equals(ExprTk::genLinkRemoverPostCond2(ca, a)) )
```

XL	

```
-- for-all link-adder operation p of c = owner(a) that adds a collection
--  of objects of c = a.type, the pre- and post-conditions of p
--  must be set accordingly
context a: Field inv CardMax :
  let x : Integer = a.asc.associate.cardMax in
  x >= 1 implies
  let c2 : Class = elementType(a.type) in
  a.owner.linkAdders()->forAll(p |  p.isAttrRef(a) implies
    p.params()->size() = 1 and let m : Parameter = p.params().any(true)
    in elementType(m.type) = c2 and
      p.preExps().exists(e |
        e.equals(ExprTk::genLinkAdderPreCond(c2, m, x))) and
      p.postExps().exists(e1, e2 |
        e1.equals(ExprTk::genLinkAdderPostCond1(a, m)) and
      e2.equals(ExprTk::genLinkAdderPostCond2(c2, a)) )
```

Appendix B. A partial DCSL specification of CourseMan written in Java

Class Student

```
@DClass(mutable=true) public class Student {
  /*** STATE SPACE **/
  @DAttr(name="id",type=Type.Integer,id=true,auto=true,mutable=false,optional=false)
  private int id;
  @DAttr(name="name",type=Type.String,mutable=true,optional=false,length=30)
  private String name;

  @DAttr(name="enrolments",type=Type.Collection)
  @DAssoc(ascName="std-has-enrols",role="std",
    ascType=AssocType.One2Many,endType=AssocEndType.One,
    associate=@Associate(type=Enrolment.class,cardMin=0,cardMax=30))
  private Collection<Enrolment> enrolments;
  private int enrolmentCount;
  private static int idCount;

  /*** BEHAVIOUR SPACE (partial) **/
  @DOpt(type=DOpt.Type.ObjectFormConstructor,
    requires="n.size() <= 30",
    effects="self.id = genId() and self.name = n")
  public Student(@AttrRef("name") String n);

  // automatically generate the next student id (if required)
  @DOpt(type=DOpt.Type.AutoAttributeValueGen,
    effects="Student::idCount = Student::idCount+1 and result = Student::idCount")
  @AttrRef("id")
  private static int genId();

  // add links (this,e) for all e in enrols
  @DOpt(type=DOpt.Type.LinkAdder,
    requires="enrolmentCount + enrols.size() <= 30",
    effects="enrolments = enrolments@pre->asSet()->union(enrols->asSet()) and
             enrolmentCount = enrolments->size()")
  @AttrRef("enrolments")
  public boolean addEnrol(Collection<Enrolment> enrols);
```

```
35
36    // add link (this,e)
37    @DOpt(type=DOpt.Type.LinkAdder,
38      requires="enrolmentCount + 1 <= 30",
39      effects="enrolments = enrolments@pre->asSet()->union(Set{e}) and
40               enrolmentCount = enrolments->size()")
41    @AttrRef("enrolments")
42    public boolean addEnrol(Enrolment e);
43
44    // remove link (this,e) for all e in enrols
45    @DOpt(type=DOpt.Type.LinkRemover,
46      requires="enrolmentCount - enrols.size() >= 0",
47      effects="enrolments = enrolments@pre->asSet() - enrols->asSet() and
48               enrolmentCount = enrolments->size()")
49    @AttrRef("enrolments")
50    public boolean removeEnrol(Collection<Enrolment> enrols)
51      throws ConstraintViolationException;
52
53    // remove link (this,e)
54    @DOpt(type=DOpt.Type.LinkRemover,
55      requires="enrolmentCount - 1 >= 0",
56      effects="enrolments = enrolments@pre->asSet() - Set{e} and
57               enrolmentCount = enrolments->size()")
58    @AttrRef("enrolments")
59    public boolean removeEnrol(Enrolment e) throws ConstraintViolationException;
60
61    // getter for name
62    @DOpt(type=DOpt.Type.Observer,effects="result = name") @AttrRef("name")
63    public String getName();
64
65    // setter for name
66    @DOpt(type=DOpt.Type.Mutator,
67    requires="n.size() <= 30", effects="self.name = n")
68    public void setName(@AttrRef("name") String n);
69  }
```

Class Enrolment

```
1   @DClass(mutable=true)
2   public class Enrolment {
3     /*** STATE SPACE (partial): other attributes (omitted) */
4     @DAttr(name="student",type=Type.Domain,optional=false)
5     @DAssoc(ascName="std-has-enrols",role="enrolments",
6       ascType=AssocType.One2Many,endType=AssocEndType.Many,
7       associate=@Associate(type=Student.class,cardMin=1,cardMax=1))
8     private Student student;
9
10    @DAttr(name="module",type=Type.Domain,optional=false)
11    @DAssoc(ascName="mod-has-enrols",role="enrolments",
12        ascType=AssocType.One2Many,endType=AssocEndType.Many,
13      associate=@Associate(type=CourseModule.class,cardMin=1,cardMax=1))
14    private CourseModule module;
15
16    /*** BEHAVIOUR SPACE (omitted) */
17  }
```

Appendix C. A detailed expressiveness comparison between DCSL and the DDD frameworks

Table C.7: Comparing the expressiveness of DCSL to AL, XL.

DCSL	AL	XL
DClass		
mutable	DomainObject.editing	–
DAttr		
unique	– $^{(i)}$	-
optional	jdo$^{(ii)}$.Column.allowsNull, (Property.optionality)	Required
mutable	Property.editing	-
id	jdo.PrimaryKey.value	jpa.Id
auto	–	– $^{(iii)}$
length	jdo.Column.length, (Property.maxLength)	jpa$^{(iv)}$.Column.length
min	–	Min$^{(v)}$.value
max	–	Max$^{(v)}$.value
DAssoc		
ascName	–	–
ascType	–	jpa.OneToMany, jpa.ManyToOne, jpa.ManyToMany
role	–	–
endType	–	–
associate.type	–	–
associate.cardMin	–	–
associate.cardMax	–	–
DOpt		
type	–	–
requires	–	–
effects	–	–
AttrRef		
value	–	–

(i) AL supports property `Property.mustSatisfy` which may be used to implement the constraint.

(ii) Java Data Objects (JDO) [40].

(iii) XL supports property `ha.Formula` that may be use to implement formula for value generation function.

(iv) Java Persistence API (JPA) [39].

(v) Bean Validator (BV) [42].

Appendix D. A detailed level of coding comparison between DCSL and the DDD frameworks

Table D.8: Comparing the max-locs of DCSL to AL, XL.

DCSL	AL	XL
Domain Class		
DClass.mutable	DomainObject.editing, autoCompleteRepository	Entity, EntityValidator.value
Domain Field		
DAttr.length, min, max	Column.allowsNull, name, *one-of* { length, scale }, jdbcType	Required, *one-of* { Column.length, scale }, PropertyValidator.value, Min.value, Max.value, SearchKey
Associative Field		
DAssoc.ascName, ascType, role, endType; associate.type, associate.cardMin, associate.cardMax	–	*one-of* { OneToMany, ManyToOne, ManyToMany}

Table D.9: Comparing the typical-locs of DCSL to AL, XL

DCSL	AL	XL
Domain Class		
DClass.mutable	DomainObject.editing, autoCompleteRepository	Entity, EntityValidator.value
Domain Field		
one-of { DAttr.length, min, max }	*one-of* { Column.length, scale }	*one-of* { Column.length, Column.scale, Min.value, Max.value }
Associative Field		
DAssoc.ascName, ascType, role, endType; associate.type, associate.cardMin, associate.cardMax	–	*one-of* { OneToMany, ManyToOne, ManyToMany}