Domain-driven design patterns: A metadata-based approach

Conference Paper · November 2016

DOI: 10.1109/RIVF.2016.7800302

3 authors:

Duc Minh Le
Swinburne University of Technology Vietnam
17 PUBLICATIONS **42** CITATIONS
SEE PROFILE

Duc-Hanh Dang
Vietnam National University, Hanoi
30 PUBLICATIONS **134** CITATIONS
SEE PROFILE

Viet Ha Nguyen
Vietnam National University, Hanoi
62 PUBLICATIONS **162** CITATIONS
SEE PROFILE

Domain-Driven Design Patterns: A Metadata-Based Approach

Duc Minh Le
Department of Software Engineering
Hanoi University, Vietnam
Email: duclm@hanu.edu.vn

Duc-Hanh Dang, Viet-Ha Nguyen
Department of Software Engineering
VNU - University of Engineering and Technology, Hanoi, Vietnam
Email: {hanhdd, hanv}@vnu.edu.vn

Abstract—**Design pattern is the most common form of object oriented software reuse. In object oriented domain driven design, a number of high-level patterns have been identified and applied for over a decade. However, no concrete design patterns for domain modelling in this method have been published in the literature. A primary challenge in defining these design patterns is how to express their form in a way that eases their application to a specific domain that uses a specific object oriented programming platform. In this work, we propose a set of concrete design patterns, whose form is expressed in a domain class modelling language (DCML). DCML is based on UML and uses implementation-aware meta-attributes to define design metadata. We extend this language with new meta-attributes to support the proposed design patterns. Further, we discuss how domain-specific examples of these patterns are translated to physical class models for automatically generating software prototypes.**

I. Introduction

The overall goal of domain-driven design (DDD) [1] is to design software (iteratively) around realistic domain model(s), which both thoroughly capture the domain requirements and are technically feasible for implementation. A core principle of object-oriented DDD [1] for achieving this goal is to use a ubiquitous language that is structured based on the domain model(s). This language helps bring together the three key stakeholders (domain expert, designer and programmer) and enables them to collaboratively build and eventually implement the domain model(s) in a target object-oriented programming language of choice. Since inception, the DDD's author has stressed the importance of using design patterns to enrich the ubiquitous language. The high-level descriptions of the patterns given in [1], [2] make use of a number of generic object-oriented design patterns ([3]).

However, our research reveals that no concrete design patterns for domain modelling in DDD have so far been published in the literature. A primary challenge in defining these design patterns is how to express their form in the way that eases their application to a specific domain and a specific object oriented programming platform (e.g. Java [4], C#/.Net [5]).

In this work, we propose to address these with a set of core concrete design patterns, which we name domain-driven design patterns (DDDPs). These patterns' form is expressed in a metadata-based domain class modelling language (DCML [6]). This language is designed based on UML and using implementation-aware meta-attributes to define design metadata. We extend this language with new meta-attributes to support the proposed DDDPs. Further, we discuss how domain-specific examples of the DDDPs are translated to physical class models, which are then used, with the help of a tool, to automatically generate software prototypes.

The rest of the paper is structured as follows. Section II discusses some background terms and motivation of the work. Section III presents our extension of DCML. Section IV presents the core DDDPs. Section V discusses tool support. Section VI summarises the related work and Section VII concludes the paper.

This work is supported by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2015.25. We wish to thank the anonymous reviewers for their useful comments.

II. Background And Motivation

In this section, we review some key background terms and explain the motivation for both the concrete design patterns and the use of DCML for expressing them. We illustrate using a course management *software domain*, named CourseMan. This domain combines the essential requirements of the software of the same name in [6] and [7]. These requirements describe a compact, yet complete, domain that supports both data and activity concepts.

According to [8] a **design pattern** is a frequently-used design abstraction, whose form "...is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship." The DDD's patterns defined in [2] do not exactly fit this definition, as their forms are described only at the conceptual (high) level.

We argue that to effectively apply these patterns require concrete design patterns, which we term domain-driven design patterns. A **domain-driven design pattern (DDDP)** is a design pattern that addresses a domain modelling problem, is described in a structured format, and whose form is a template model that is expressed in a well-defined modelling language.

Figure 1 shows the initial UML domain model of CourseMan, which contains six *domain classes*. Classes Student, Address, Enrolment, CourseModule, and SClass represent five data concepts of the domain. Together, these capture the requirements about Students, their associated Addresses, their Enrolments in one or more CourseModules, and their groupings into one or more SClasses (abbreviated for Student Classes). Class EnrolmentMgmt models the overall course registration activity, which involves registering Students into CourseModules and organising Students into SClasses.

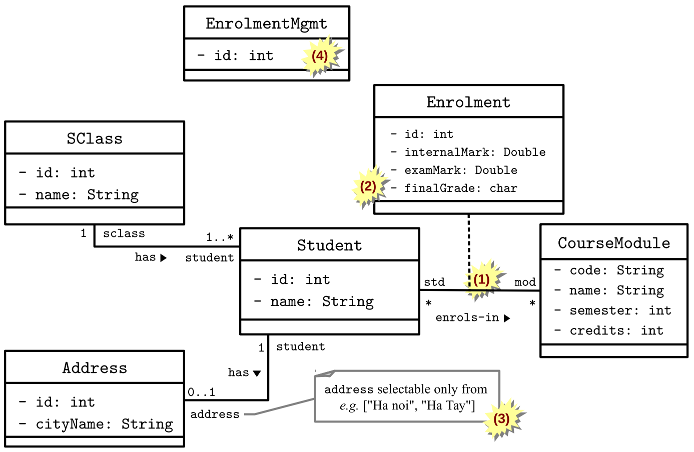


Fig. 1: The initial UML domain model of COURSEMAN

There are four non-trivial, frequently-recurring concrete design problems that are marked with the labels (1)-(4) in the domain model of Fig. 1. Solving and documenting these problems in the form of DDDPs will allow these patterns to easily be reused to create the domain models of other software domains that face the same problems. The four problems are briefly stated as follow (discussed in detail in Section IV):

(1) How to design a many-many association, such as that between Student and CourseModule, such that it is aware of the intermediate class (class Enrolment in this example) that normalises it?
(2) How to design such multi-source derived attributes as Enrolment.finalGrade (this attribute is derived from two other attributes: internalMark and examMark)?
(3) How to design an association role, such as address, in terms of a referenced attribute (e.g. Student.address) such that it is selectable only from a subset of linked domain objects (Addresses in this example)?
(4) How to design such activities as EnrolmentMgmt as a domain class, such that they can be modelled in the domain model with some form of relationships to the relevant data classes (e.g. SClass and Student)?

A closer look at these problems reveal that they all require enriching the standard UML constructs with design metadata that are beyond their current capabilities to support. More specifically, problem (1) requires linking an association to a class, (2) requires linking different attributes of the same class, (3) requires adding some form of query to the definition of an attribute, and (4) requires linking classes that represent data and activity concepts.

This leads us to an idea of using the domain class modelling language (DCML) [6] to express both the domain model and the DDDP's form. DCML is based on UML but supports the use of meta-attributes to define design metadata. This language has the following key features that are suitable not only for expressing the domain model of DDD but for addressing the challenge stated in Section I concerning the DDDP's form:

- *designer-friendly*: the meta-attributes are represented on a UML class diagram using a structured note box
- *automatic behaviour generation*: behavioural specification can be generated from the attribute specification
- *automatic code generation*: the meta-attributes are translated automatically to code using mapping rules ([6])
- *ease of support for domain model prototyping*: the physical class model of the domain model is processed by a tool (discussed in Section V) to generate an interactive software for rapidly prototyping the domain model

III. EXTENDING DCML FOR DDDPS

According to [6], DCML is standard UML ([9]) augmented with meta-attributes. We define in this section the key concepts of DCML and an extension for DDDPs. Examples of these concepts will be given shortly in Section IV.

Semantically, a **meta-attribute** is a named set of properties that characterise one of the following UML constructs: class, attribute, association, association end, and operation. A property is a (name, value) pair. Syntactically, a meta-attribute is structurally written in a form consisting of its name followed by a sequence of its properties, written in the form of (name, value) pairs. The ordering of properties is insignificant.

A **meta-attribute is assigned** either to a standard UML element or to the value of a property of another meta-attribute. In both cases, the assignment results in properties of the meta-attribute being assigned suitable values.

We thus consider a **domain model** as being consisted of two sub-models: **standard model** and **metadata model**. The former contains only the standard UML elements, while the latter contains only the meta-attribute assignments. Accordingly, we represent the *DDDP form* as a **template domain model** – a 'parameterised' domain model, in which elements of the standard model are named after the generic roles that they play (rather than using domain-specific terms). Further, in order to define the template domain models of the core DDDPs in this paper, we propose to extend DCML with three meta-attributes: QueryDef, AttribExp, and AssocEnd.

A. Meta-attributes QueryDef, AttribExp

These meta-attributes are used to define an object-based query that identifies the set of objects allowable for an attribute. First, meta-attribute QueryDef, which is assigned to attribute, defines the query. It has two properties: clazz, attributeExps. Property clazz specifies the name of the domain class whose objects are subject of the query (i.e. appearing in the query's result set). Property attributeExps specifies one or more attribute-specific expressions of the query. Each expression is typed AttribExp.

Meta-attribute AttribExp has three properties: attrib, op, value. Property attrib specifies the name of an attribute of the query's class, whose value is evaluated against a function whose abstract form is *op value* (i.e. constructed from the values of the other two properties).

B. Meta-attribute AssocEnd

Semantically, meta-attribute AssocEnd is similar to DAssocEnd in [6] in that both specify properties of an association end. Syntactically, however, AssocEnd is easier to write and read than DAssocEnd, because it uses the look-here notation (DAssocEnd uses the look-cross notation). In [10], both notations are generalised and applied to UML.

AssocEnd has one core property named role (role name of the referenced association end). We will introduce other properties shortly in Section IV. An array of AssocEnds is assigned to property DAssoc.assocEnds (DAssoc captures metadata about association [6]).

IV. The Four Core DDDPs

We present in this section the four core DDDPs that address the problems stated in Section II. We begin with the definition of the pattern description form.

A. Pattern Description Form

We use the following extended pattern description form of [8] to describe DDDPs: $\langle context, form, example \rangle$. Section *context* describes the pattern's context (intent and motivation). Section *form* describes the solution in terms of a template domain model. Section *example* gives an example domain model and code example that implements the pattern. Due to space constraint, however, the code examples of the patterns are listed in [11] but not this paper.

Also due to space constraint, we will omit from the metadata model of both the template domain model and example domain model the elements (most noticeably metadata assignments and domain operations) that are discussed in [6] as being standard for them. When there is a need in some cases to emphasise the type of certain elements, we will use a simple note box or a reduced metadata note box instead of a full-featured metadata one.

To ease notation, in both the template domain models and the example ones, we will generally assume that classes, attributes and operations are domain classes, attributes, and operations (respectively), as defined in [6].

B. Many-Many Association Normaliser

1) Context:

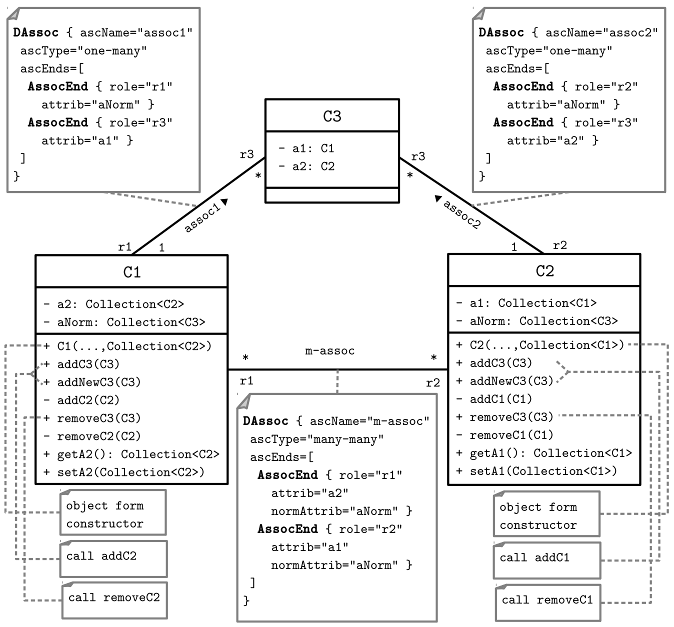
a) Intent: To normalise the definition of many-many associations to meet the requirements of the application's infrastructure (e.g. data storage) without affecting the user's manipulation of these associations.

b) Motivation: While the presence of many-many associations is normal in the domain model, they often cause problems in the infrastructure layer of the application that need to handle of the objects that participate in such associations. A common problem is when the infrastructure layer uses a relational database to store objects and this database does not inherently support many-many associations.

A typical solution is to normalise a many-many association into a pair of one-many associations, that associate the two participating classes indirectly via an intermediate class. A limitation of this solution, however, is that it would lead to the user having to work with the intermediate class rather than directly with the association.

This raises the need for a better design solution that helps insulate the user from details of the intermediate class and provides (s)he with a mechanism to directly manipulate the association links of a many-many association.

2) Form: The following figure shows the template domain model of the pattern. The standard model consists of three classes C1, C2, and C3. The many-many association associates C1 and C2; C3 is the intermediate class that normalises this association. Class C1 has two attributes a2 and aNorm, C2 has two attributes a1 and aNorm, and C3 has two attributes a1 and a2. Attribute aNorm is the normaliser attribute. Classes C1 and C2 each has an object form constructor and two private adder and remover operations that add and (resp.) remove objects of the other class. The constructor of one class has a Collection-typed parameter that is used to input objects of the other class. The adder and remover operations are called by the link-adder and link-remover operations of the normaliser attribute.



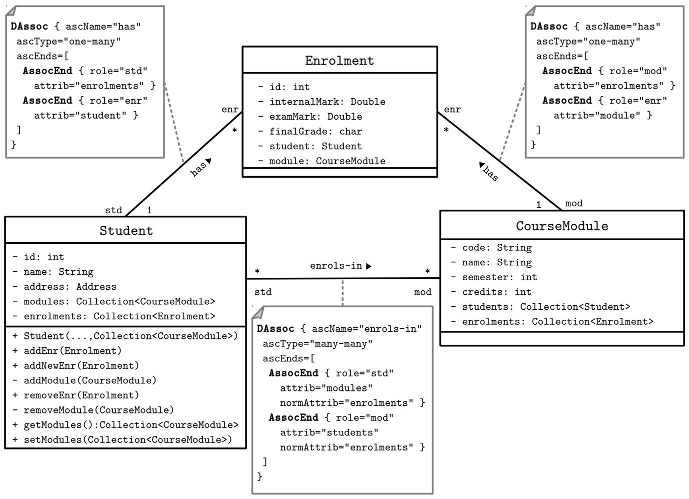
Note that due to the symmetrical nature of many-many association, the above operation sets can be defined for either C1 or C2 (not both), depending on which class is used for user input. We will demonstrate this shortly in the example.

In the metadata model, the assignments of DAssoc to the three associations define the normalisation scheme. Note, in particular, that the two AssocEnds of the many-many association have the property normAttr set to name of the normaliser attribute of the two associated classes.

3) Example: The following figure shows how the pattern is applied to solve problem (1) of CourseMan. In this example:

- C1 = Student, C2 = CourseModule, C3 = Enrolment

Note that class CourseModule does not include the operations set. This is because class Student is chosen for capturing user data about student registration into course modules.



C. Multi-Source Derived Attribute

1) Context:

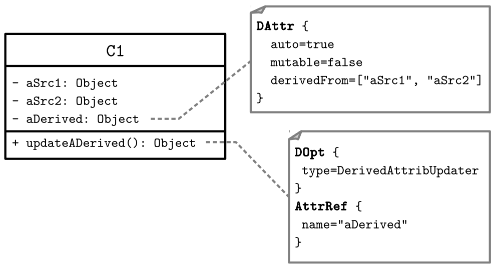
a) Intent: To explicitly declare that a class's attribute (derived attribute) is derived from some other attributes (source attributes) of that class so as to provide clients with control over when to update the value of the derived attribute.

b) Motivation: A simple design technique for derived attributes is to define them as private attributes and update their values when those of the source attributes are changed. However, by hiding the value update of the derived attribute from the clients (of the owner class) this solution suffers from two main limitations. First, it does not provide clients with control over when a derived attribute's value is actually updated. Second, it does not provide clients with access to the former value of a derived attribute prior to the update.

A better design solution that overcomes these limitations is to define a public derived-attribute-updater operation for each derived attribute and to explicitly define the connection between the derived attribute and its source attributes so that, especially in the case of multi-source attributes, clients can wait until all the source attributes have been updated before calling the updater operation to update the derived attribute.

2) Form: The following figure shows the template domain model of the pattern. The standard model consists of a class C1 that is designed with one derived attribute (aDerived), the source attributes (aSrc1, aSrc2) of this attribute, and the public, derived-attribute-updater operation named updateADerived. This operation has an empty parameter list and has a return type suitable to the declared type of attribute aDerived. The empty parameter list is important to ensure that this operation does not violate the mutable constraint of the derived attribute (specified below), while still providing clients with the ability to invoke this operation when needed.

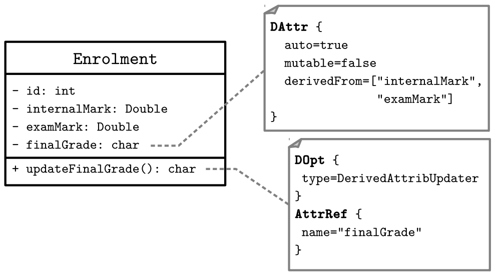
In the metadata model, the connection between the derived attribute and the source attributes is represented by the assignment of DAttr to the attribute aDerived in which: auto=true, mutable=false, and property derivedFrom is set to an array containing the names of the source attributes. Further, the public mutator operation is assigned the meta-attribute DOpt



and AttrRef. Property DOpt.type is set to the type constant DerivedAttribUpdater (which means an operation that updates a derived attribute), while property AttrRef.name is set to the name of the derived attribute.

3) Example: The following figure shows how the pattern is applied to solve problem (2) of CourseMan. In this example:

- C1 = Enrolment



D. Data Source Attribute

1) Context:

a) Intent: To define that a class's attribute (data source attribute) is to serve as a view over the domain objects of some other related classes, and this view is needed to provide run-time data range for other attributes (called bounded attributes) of the class.

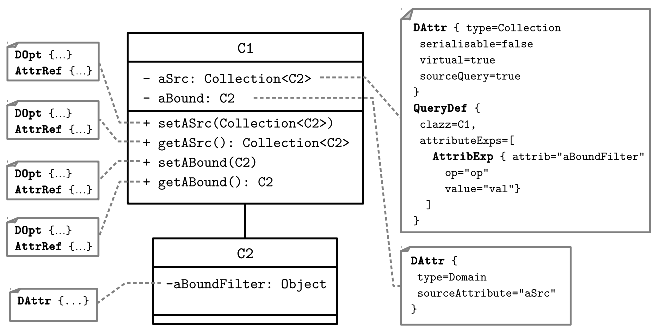
b) Motivation: Existing design techniques address the requirements of data source attribute by realising the attribute not in the class but in the UI, typically using a bounded data field to represent the attribute and defining the view query as part of this field.

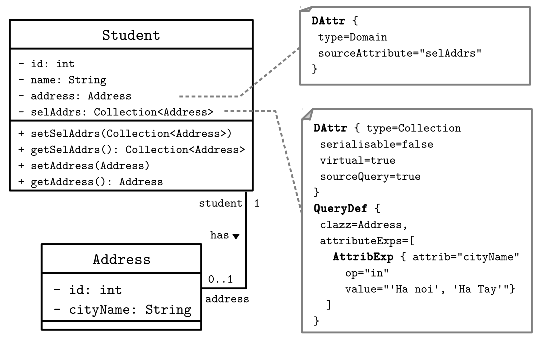
This solution has three main limitations. First, it creates unnecessary coupling between the UI and the model of an application. Second, by bundling together the definition and use of a data source attribute, it does not make it possible to reuse the definition for different clients. Third, by assuming that the query is known at design time, the design is not applicable if the actual data is only available at run-time, especially when these data are prepared by other components.

A better solution that overcomes these limitations is to separate the definition from the use of the data source attribute and to place the definition directly in the class.

2) Form: The top-part of the following figure shows the template domain model of the pattern. The standard model consists of two classes C1 and C2. Class C1 has a bound attribute (aBound) that is typed C2, a data source attribute (aSrc) typed Collection⟨C2⟩, and setters and getters for these two attributes.

Class C2 is designed with an attribute named aBoundFilter, whose value is used to filter a sub-set of C2 objects that will actually be populated into the attribute aSrc.





First in the metadata model is the assignment of DAttr to attribute aBound. In this assignment, property type=Domain because aBound is a reference type; property sourceAttribute makes explicit the binding of aBound to the attribute aSrc.

Second, the metadata assignments of DAttr and QueryDef to the attribute aSrc make explicit the fact that this attribute is a data source attribute whose data query is defined by QueryDef. Specifically, we have DAttr.type=Collection, serialisable=false, because aSrc is a collection-typed attribute. DAttr.virtual=false because aSrc is not part of the object state. DAttr.sourceQuery=true makes explicit the presence of the QueryDef assignment. The query defined by QueryDef is used to identify the C2 objects the values of whose aBoundFilter attribute satisfy the expression (op val), for some operator op and some value val. More than one attribute expressions can be set using the property QueryDef.attributeExps.

Third, the metadata assignments of DOpt and AttrRef to the four setter and getter operations of class C1 make explicit the behaviour types of these operations. Details of these assignments are omitted from the figure as they are considered standard assignments. Fourth, the assignment of DAttr to attribute aBoundFilter of C2 is necessary to turn this attribute into a domain attribute, and thus can appear in QueryDef. The details of this assignment are also omitted from the figure.

Note that the metadata assignment of QueryDef may be omitted either if no design-time query is known or if attribute aSrc refers to all the C2 objects (and thus making the query definition unnecessary). In the former case, the setter operation setASrc is used by clients to populate the run-time data for attribute aSrc.

3) Example: The bottom-part of the above figure shows how the pattern is applied to solve problem (3) of COURSEMAN. In this example:

- C1 = Student, C2 = Address, and attribute Student.address realises the role address

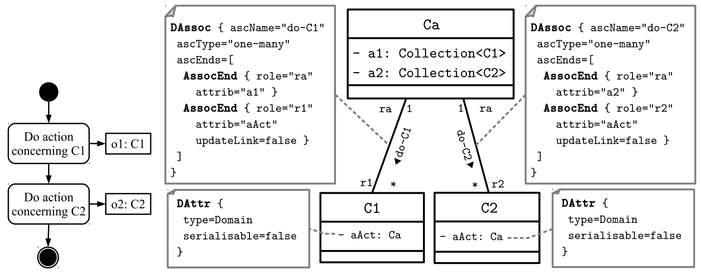
E. Activity Class

1) Context:

a) Intent: To model an activity (defined in the UML specification ([9])) as a domain class.

b) Motivation: A class that models an activitiy is called activity class. The key question is what constitute the structural and behavioural features of an activity class? Our study of the contemporary domain-driven software frameworks ([12], [13]) reveals that there is no uniform answer to this question. The design solutions proposed by these frameworks differ fundamentally in how they view the relationship between an activity class and the data classes (called relevant data classes), whose data are used and/or manipulated by the activity.

We propose that activity class be modelled as a domain class. A key benefit of this is a uniform representation for both data and activity concepts. Such representation increases productivity and modularity, while reducing maintenance.

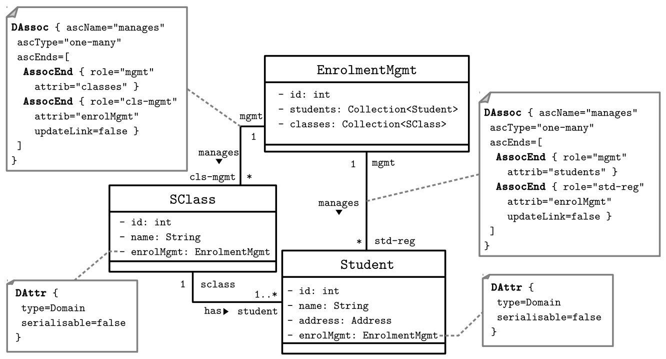


2) Form: The right hand side of the above figure shows the template domain model of the pattern. The standard model consists of three classes Ca, C1, and C2. Class Ca is the activity class for the activity whose diagram is shown on the left hand side of the figure. Class Ca maintains two associations with two relevant data classes C1 and C2. The objects of these two classes are manipulated as part of performing the two actions (shown in the activity diagram) concerning these two classes. In this work, we focus on the basic structure of the activity diagram which includes no fork, join, or decision nodes. Support for these constructs will be investigated as part of future work.

In the metadata model, the assignments of DAssoc to the two associations make explicit the fact that the two associations are one-many and, more importantly, that updateLink=false, which means objects of C1 and C2 are to be managed independently from those of Ca. This also means that links to Ca objects will not be created/updated for objects of C1 and C2. The two attributes aAct of C1 and C2 are assigned with a standard configuration of DAttr that has type=Domain and serialisable=false.

3) Example: The following figure shows how the pattern is applied to solve problem (4) of COURSEMAN. In this example:

- Ca = EnrolmentMgmt, C1 = SClass, C2 = Student



```
DAssoc { ascName="manages"
 ascType="one-many"
 ascEnds=[
  AssocEnd { role="mgmt"
    attrib="classes" }
  AssocEnd { role="cls-mgmt"
    attrib="enrolMgmt"
    updateLink=false }
 ]
}
```

```
DAssoc { ascName="manages"
 ascType="one-many"
 ascEnds=[
  AssocEnd { role="mgmt"
    attrib="students" }
  AssocEnd { role="std-reg"
    attrib="enrolMgmt"
    updateLink=false }
 ]
}
```

```
DAttr {
 type=Domain
 serialisable=false
}
```

```
DAttr {
 type=Domain
 serialisable=false
}
```

V. Tool Support

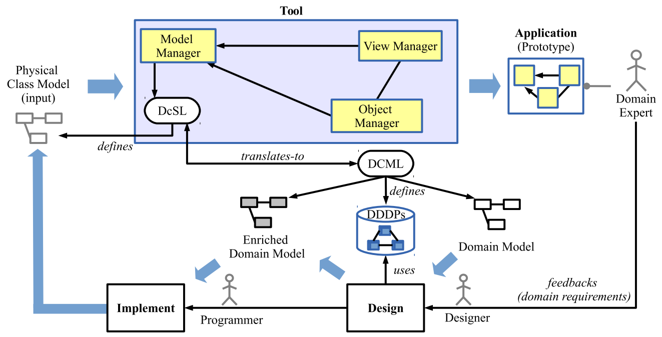


Fig. 3: Prototyping the domain models using DDDPs

The DDDPs are used to enrich the domain models, which are then used, with the help of a tool, to generate sofware prototypes. The domain models are first translated to physical class models, which are then fed into the tool to automatically generate the prototypes. The entire process is captured in Fig. 3, which is updated from the tool architecture in [6]. A Java implementation of this tool (named jDomainApp [7]) was used to generate the CourseMan prototypes that had been enriched by the core DDDPs. The user interfaces of these prototypes are presented in [11].

The top part of Fig. 3 is a summary of the same part of the architecture in [6]. In this part, the physical class model is expressed in an internal DSL (named DcSL in [6]).

The bottom part of the figure, which is currently performed manually by the designer/programmer, is revised from the one in [6] to include DCML and a (logical) collection of DDDPs. DDDPs are picked from this collection to enrich the domain models. The double-arrowed line connecting DCML and DcSL means that these languages are translatable to one another. Our plan is to automate this bottom part into the tool.

VI. Related Work

The authoritative work in [3] systematically defines a set of generic design patterns and categorised these as being creational, structural, or behavioural. Some of these patterns were used in [1] to discuss the high-level patterns of DDD ([2]). However, both [1] and [2] do not use a specific modelling language for expressing the pattern form.

Our work improves DDD with a set of concrete DDDPs and a modelling language for expressing the pattern form.

With regards to pattern form language, a common base language is UML [9]. The work in [3], [8] (as well as our work) expresses the pattern form at the model level, while others (most recently [14]) proposed to do so at the meta-model level. Our work is novel in the use of DCML as the pattern form language. We extended DCML with three new meta-attributes to express the form of the four core DDDPs. Further extension is possible to support the requirements of other DDDPs that would be identified in the future.

VII. Conclusion

In this paper, we presented a proposal for domain-driven design patterns that realise the high-level patterns of the DDD method. We argued why DDDPs are necessary and discussed the case for four core DDDPs that resolve four non-trivial, frequently-recurring design problems. We proposed to use a domain class modelling language (DCML) for expressing the DDDPs' form. This language is based on UML and uses meta-attributes to define design metadata. It has a number of key features that are suitable for expressing not only the domain model of DDD in general but the DDDP's form. We extended DCML with new meta-attributes to support the DDDPs and described each core DDDP in detail. We showed, using an extended tool architecture of a previous work, how domain-specific examples of the DDDPs are translated to physical class models for automatically generating software prototypes.

We have used the core DDDPs in developing several practical software (one of which was used as the motivating example in this paper). Our plan for future work includes improving the aforementioned tool to incorporate the support for the definition of DCML, domain models, and DDDPs.

References

[1] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[2] ——, *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, LLC, Sep. 2014.

[3] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Reading, Mass: Addison-Wesley Professional, Nov. 1994.

[4] K. Arnold *et al.*, *The Java programming language*, 4th ed. Addison-wesley Reading, 2005, vol. 2.

[5] A. Hejlsberg *et al.*, *The C# Programming Language*, 4th ed. Addison Wesley, 2010.

[6] D. M. Le, "Domain-Driven Design Using Meta-Attributes: A DSL-Based Approach," Hanoi University, Hanoi, Tech. Rep., 2016.

[7] ——, "jDomainApp: A Domain-Driven Application Development Framework in Java," Hanoi University, Tech. Rep., 2016.

[8] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *Theory and Practice of Object Systems*, vol. 2, no. 1, pp. 3–13, Jan. 1996.

[9] OMG, "Unified modeling specification: Superstructure version 2.4.1," OMG, Tech. Rep., 2011.

[10] M. Boyd and P. McBrien, "Comparing and Transforming Between Data Models Via an Intermediate Hypergraph Data Model," in *Journal on Data Semantics IV*. Springer, 2005, no. 3730, pp. 69–109.

[11] D. M. Le, "Domain-Driven Design Patterns: A Metadata-Based Approach," Hanoi University, Hanoi, Tech. Rep., 2016.

[12] Apache-S.F, "Apache Isis," 2016. [Online]. Available: http://isis.apache.org/

[13] P. Javier, "OpenXava Documentation," 2016. [Online]. Available: http://openxava.wikispaces.com

[14] D. Kim, "The role-based metamodeling language for specifying design patterns," *Design Pattern Formalization Techniques*, pp. 183–205, 2007.