jDomainApp: A Module-Based Domain-Driven Software Framework

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

Duc-Hanh Dang
Department of Software Engineering,
VNU University of Engineering and Technology, Vietnam National University, Hanoi
Hanoi, Vietnam
hanhdd@vnu.edu.vn

Ha Thanh Vu
Software Engineer,
MobiFone Corporation
Hanoi, Vietnam
ha.vuthanh@mobifone.vn

ABSTRACT

Object-oriented domain-driven design (DDD) has been advocated to be the most common form of DDD, thanks to the popularity of object-oriented development methodologies and languages. Although the DDD method prescribes a set of design patterns for the domain model, it provides no languages or tools that realise these patterns. There have been several software frameworks developed to address this gap. However, these frameworks have not tackled two important software construction issues: generative, module-based software construction and development environment integration. In this paper, we propose a framework, named jDomainApp, and an Eclipse IDE plugin to address these issues. In particular, we extend our recent works on DDD to propose a software configuration language that expresses the software configuration, needed to automatically generate software from a set of modules. The modules are automatically generated using a module configuration language that we defined in a previous work. We demonstrate the framework and plug-in using a real-world software example. Further, we evaluate the performance of software construction to show that it is scalable to handle large software.

CCS CONCEPTS

• Software and its engineering → Object oriented frameworks; **System modeling languages**; **Domain specific languages**; **Integrated and visual development environments**.

KEYWORDS

domain-driven design, software framework, system modelling, domain-specific language, development environment.

ACM Reference Format:
Duc Minh Le, Duc-Hanh Dang, and Ha Thanh Vu. 2019. jDomainApp: A Module-Based Domain-Driven Software Framework. In *The Tenth International Symposium on Information and Communication Technology (SoICT 2019), December 4–6, 2019, Hanoi - Ha Long Bay, Vietnam, Viet Nam.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3368926.3369657

1 INTRODUCTION

Domain-driven design (DDD) [4, 14, 21] has been one of the hottest software engineering topics over the past two decades. Since the rise of high-level object-oriented programming languages (OOPLs), such as Java [6] and C# [7], object-oriented DDD has emerged to become a most common form of DDD. The basic DDD's philosophy is this: the core (a.k.a "heart") of software is the domain model and, thus, effectively constructing this model is a central issue. For this, Evans [4, 5] prescribes a set of design patterns. However, he does not define any specific languages to realise the patterns nor does he provide any tools that support the method. More recently, a number of software frameworks, most noticably ApacheIsIs [3] and OpenXava [18], have been developed to bridge this gap. A common feature of this framework is that they provide reusable components for developers to construct the domain model with sufficient details so that it can be used as input to automatically generate a GUI-based software. However, these frameworks have not tackled two important software construction issues: (i) generative, module-based software construction and (ii) development environment (DevEnv) integration. In this paper, we propose a novel solution to address these issues. For the first issue, we propose a software framework, named jDomainApp, which we have partially developed in recent works [12, 13] and will enhance further in this paper. Our enhancement consists in a generative software construction method that includes a language, named sccl, for expressing the configuration of software class. We use this configuration to automatically generate a software from a set of user-specified modules. The modules themselves are automatically generated using a module configuration language that we defined in a previous work [13]. For the second issue, we propose an Eclipse plugin that modularises the integration of jDomainApp into the Eclipse IDE. We demonstrate jDomainApp and the Eclipse plugin-in using a real-world software development example. Further, we evaluate software construction to show that it is scalable to handle large software.

The rest of the paper is structured as follows. Section 2 explains our design method for jDomainApp and describes a running example. Section 3 presents a high-level architecture of jDomainApp. Section 4 discusses software generation. Section 5 explains the Eclipse plugin. Section 6 presents the evaluation result. Section 7 reviews the related work and Section 8 concludes the paper.

2 DESIGN METHOD AND EXAMPLE

Our goal is to develop jDomainApp as a grey-box object-oriented software framework [19], that tackles the key software development issues that underlie the DDD method [4]. To investigate these issues, we group them into three subdomains: (*i*) domain class construction (which forms the core of domain modelling in DDD), (*ii*) software module construction and (*iii*) software construction (from modules). jDomainApp is a grey-box in that it enables us to model each subdomain using a combination of concrete and abstract classes. A *client software* can use the abstract classes to extend the functionality of the framework in order to express the domain-specific requirements.

Our approach is to treat software as a set of modules and module as a 'little' software, that is constructed based the MVC [10] architectural model. To achieve software-level reuse, we adapted the *generative software development* (GSD) approach [2] to construct the module. Our plan is to apply the same approach in this paper to construct the software itself. In principle, a *software* is considered in GSD as an instance (a.k.a variant) of a class of software that share similar features. The aim of GSD is to construct the software class (a.k.a software family [2]) so that it both contains sufficent base components and allows variation points to be defined for the software instances as they are implemented for specific domains. Software frameworks are a natural fit for use as or a platform on which to develop the base components. Since not every feature of a base component is needed in every application domain, to develop a software from the base components requires a language that can express which component features are actually needed for a domain. We call this the *configuration language* and treat it as a type of *domain-specific language* (DSL) [9].

We are particularly interested in a type of DSL named *annotation-based DSL* (aDSL) [15], which is an internal DSL whose syntax is expressed directly in a host target object-oriented programming language (OOPL) that supports annotation. We argued in [12] that Java [17] and C# [7] are suitable host OOPLs. For subdomain (*i*), we defined an aDSL, named DCSL [12], to construct each domain class in the domain model. For subdomain (*ii*), we defined another aDSL, named MCCL [13], to construct *module configuration classes* (MCCs), that are needed to automatically generate software modules directly from each domain class. In the next section, we will give more details concerning our solutions for these two subdomains. We will also present an overview of our approach that we use in the remainder of the paper to tackle the third subdomain.

Software Example: CourseMan

To illustrate the concepts presented in this paper, we use a slightly simplified example software, named CourseMan [12, 13], that is developed for the course management domain. The core *domain model* of CourseMan is shown in Figure 1. The model is expressed by the following fundamental UML [16] meta-concepts: class, attribute, operation, association, association class and generalisation. For brevity, we exclude operations from the class boxes. Class `Student` represents the domain concept Student[1]. Class `CourseModule` represents the `CourseModules`[2] that are offered to students. Class

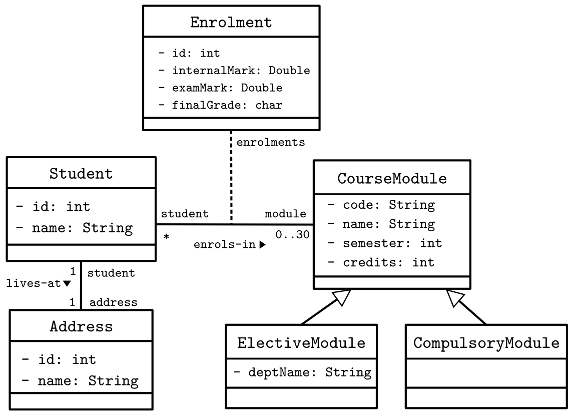


Figure 1: The core CourseMan domain model.

`Address` represents addresses where `Students` live while undertaking their studies. The study arrangement for each `Student` involves enrolling his/her into one or more `CourseModules`. This many-many association is resolved by an association class, named `Enrolment`, which contains additional information about the aggregate marks and the final grade.

3 ARCHITECTURE OVERVIEW

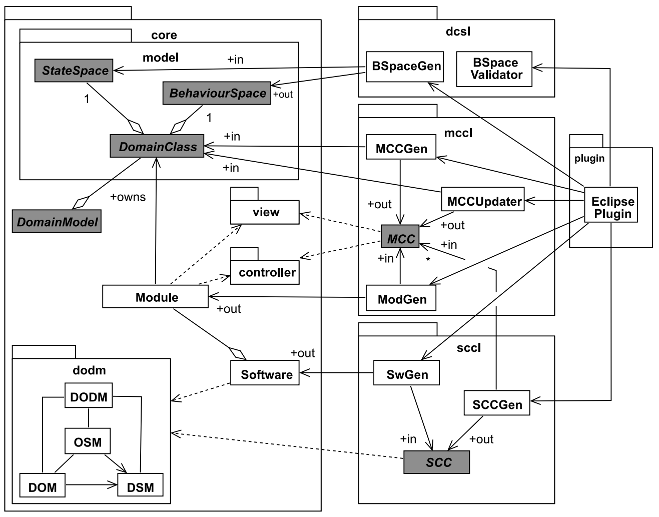


Figure 2: The high-level architecture of jDomainApp.

Figure 2 is a UML class diagram that shows the high-level design architecture of jDomainApp. We organise the architecture components into three groups: *core*, *modules*, and *integration*. The core group is represented by the package named `core` in the figure. The modules group includes the modules that extend the core functionality. In this paper, we focus on three essential modules: `dcsl`, `mccl` and `sccl`. These modules are represented by three packages in the middle column of the figure. The integration group includes components that help ease the integration jDomainApp into a target DevEnv. In this paper, we focus on the Eclipse DevEnv and introduce for this group the `plugin` package shown on the far right of

[1]we use `fixed font` for model elements and normal font for concepts.
[2]we use class/concept name as countable noun to identify instances.

figure. The abstract classes that must be realised by a client software are depicted in the figure by grey-filled rectangles.

Among the architectural components, function `BSpaceValidator` (package: `dcsl`) and the two packages `sccl` and `plugin` and are the new contributions of this work. We will discuss these in the remainder of the paper. Other components were developed as part of previous works and will be described briefly in this section.

3.1 Domain Model and DCSL

Abstract class `DomainModel` represents the domain model, which consists of a set of domain classes. Abstract class `DomainClass` represents the meta-class from which all domain classes are created. For example, the six classes in Figure 1 are derived from `DomainClass`. `DomainClass` consists of two parts: state space (class: `StateSpace`) and behaviour space (class: `BehaviourSpace`). *Behaviour space* is a design fragment of a domain class that consists of the domain methods that operate on the domain fields of the class. These fields constitute what we term the *state space*. We express `DomainClass`, `StateSpace` and `BehaviourSpace` by an aDSL named DCSL [12]. Package `dcsl` in the architecture is named after this language. To help automate the design of domain classes, we define in `dcsl` a function `BSpaceGen`: `StateSpace` ⟶ `BehaviourSpace`. This is a generator function (see [12]) that takes as input a domain class's state space and generates as output its behaviour space.

3.2 Module and MCCL

In designing jDomainApp, we adopted the Model-View-Controller (MVC) architecture [10, 20] and applied it to each software module. A module is an instance of a *module class* [11, 13], which is structured from three component classes: a domain class, a view class and a controller class. The latter two classes are parameterised from two template classes `View` and `Controller`, such that the template parameter of each of these is bound to the domain class. We say that the module class *owns* its domain class. In Fig. 2, class `Module` represents the meta-class from which all the module classes are created. The two packages `view` and `controller` contain components that make up the structures of `View` and `Controller`. The modules of a software communicate with each other using events, the arguments of which carry domain objects of interest. A module M_A can communicate with any modules whose domain classes have an association with M_A's domain class in the domain model.

To attain module-level reuse, we define another aDSL, named MCCL [11, 13], to express the MCCs. An MCC captures a module class's configuration, which references the components contained in the two packages `view` and `controller` of the architecture. The set of MCCs of a software form an *MCC model* of that software. Package `mccl` (named after MCCL) includes three functions: (*i*) `MCCGen`: `DomainClass` ⟶ `MCC`: generates an MCC from a domain class; (*ii*) `MCCUpdater`: `DomainClass` × `MCCModel` × `String` ⟶ `MCCModel`: updates an `MCCModel` when a `DomainClass` owned by one of the modules is changed. The changes are specified in a (`String`) structured text as part of the input; and (*iii*) `ModGen`: `MCC` ⟶ `Module`: generates a module class from an MCC.

To ease construction, we divide software modules into three types. The first type consists of a single special module called the *main module*. Unlike other modules, the view component of the main module describes the overall structure of the software GUI. This structure includes a container for the views of all non-main modules. In addition, it specifies the structures of the application menu and tool bar. These provide commands for the user to interact with each non-main module. The second module type is *functional module*, which are modules that are constructed from the domain model. These modules provide the domain-specific functionality. The third module type is *system module*, which provide technical functionality that is shared across different software. This functionality is similar to that provided by the infrastructure layer of the DDD's software architecture [4]. For example, we provide an essential system module, named *information module*, which captures information about the owner organisation of the software.

3.3 Software and DODM

We introduced in [12] the class `Software` to represent software and structured it in terms of a set of modules. To help effectively manage the storage of domain objects that are created by a software, we introduce a component named *domain-driven object management* (DODM). Class `DODM` is located in the package `dodm` and composed of three separate but inter-dependent classes: *domain schema manager* (DSM), *domain object manager* (DOM) and *object store manager* (OSM). In principle, DSM defines a repository of all the domain classes in the domain model, DOM maintains the object pool of each of class and OSM defines an interface to the underlying data source that is used to store objects. As shown in Fig. 2, DSM is used by the other two classes to provide information about the domain classes. DOM uses DSM to analyse a domain class's structure and to manipulate its objects; it uses OSM to store the objects. OSM uses the domain class structure it obtains from DSM to initialise the class store (e.g. a relation) in the data source (e.g. a relational database). It uses DOM to store and retrieve objects to and from their object pool. OSM is designed to support different types of data source technologies. As of this writing, it provides implementations for two commonly-used relational database management systems (RDBMSs), namely Java DB (provided by Java) and PostgreSQL[3]. PostgreSQL is a powerful open source RDBMS and is, thus, used as the default RDBMS in jDomainApp. OSM uses the standard JDBC[4] protocol to connect to these RDBMSs. A client software can easily support other data source technologies by implementing the abstract classes that are provided with OSM.

In addition, client software can provide their own implementations of DSM, DOM and OSM by subclassing these three classes. The customised classes can be specified in the software configuration, the details of which will be explained in the next section.

4 SOFTWARE GENERATION

We discuss in this section the first contribution of this work concerning the component `sccl` in the architecture. The aim is to construct a software from its modules with as much automation as possible. Based on a key observation that software is a special module that acts as the execution container for other modules, we apply the generative approach, similar to the one used to design module class, to construct software. We first define an aDSL, named

[3] http://www.postgresql.org
[4] http://www.oracle.com/database/technologies/appdev/jdbc.html

sccl, to express the software configuration classes (SCCs). We then define a function, named SCCGen, to generate an SCC and another function, named SwGen, to generate the software from an SCC.

4.1 Software Configuration Class Language

A software is an instance of a software class, which is constructed from a set of module classes and DODM. To automate software construction, we need software configuration. More generally, propose a *software configuration class language* (SCCL) to express *software configuration class* (SCC).

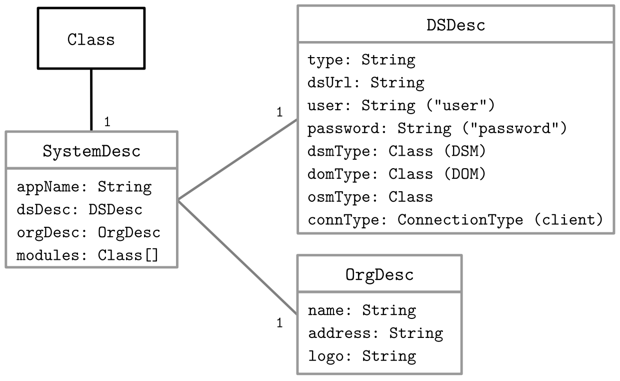


Figure 3: The abstract syntax of sccl.

Figure 3 shows an UML class model that expresses the abstract syntax of sccl. The grey-coloured boxes and association lines express the annotation concepts and associations between them. An annotation property is represented by a class field. For brevity, we add a default value (enclosed within brackets '()') next to each non-mandatory annotation property. The figure shows that sccl consists of three annotations. Annotation SystemDesc is attached to Class to form the structure of an SCC. It defines the overall software configuration and contains four properties. Two of these (desDesc, orgDesc) reference the other two annotations. Annotation DSDesc defines the configuration of the data source that is used to store domain objects. It consists of 8 properties, the first four of which are used to construct a standard JDBC's connection to the data source. The other four properties specify the types of three DODM components (DSM, DOM, OSM) and the data source connection type. To ease configuration, default values are specified for user, password, dsmType, domType and connType. Property osmType is specified based on property type.

Listing 1: An SCC of the CourseMan software

```
@SystemDesc(appName="Courseman",
 dsDesc=@DSDesc(type="postgresql",
   dsUrl="http://localhost:5432/courseman",
   user="admin", password="password",
   dsmType=DSM.class, domType=DOM.class,
   osmType=OSM.class, connType=Client),
 orgDesc=@OrgDesc(name="Faculty of IT",
   address="Km9 Nguyen Trai, Thanh Xuan",
   logo="fit.png", url="http://fit.hanu.edu.vn"),
 modules={ ModMain.class, ModAddress.class,
   ModCourseModule.class, ModEnrolment.class,
   ModStudent.class })
public class SCC1 { // empty
}
```

We use the Java's textual syntax as the concrete syntax for sccl. To illustrate, let us consider an SCC of the CourseMan software in Listing 1. The listing shows that the SCC is defined by the class SCC1. The SystemDesc assignment defines the data source to be a PostgreSQL relational database named "courseman". The SystemDesc element also defines the client organisation to be the Faculty of IT that is located at the web address http://fit.hanu.edu.vn. In addition, it specifies that the set of MCCs consists of a main module ModMain and four domain modules that own the following domain classes: Address, CourseModule, Enrolment and Student.

4.2 SCC Generator

We present in Alg. 1 the algorithm for the generator function named SCCGen. This function basically takes as input an array of MCCs (M), the software name (N) and an (optional) id label (L), and generates as output an SCC (s) of the software. If specified, L is used to create a unique SCC, which is useful to construct a different software variant. To ease comprehesion, we insert comments at the key locations in the algorithm. Notationwise, we adapted the set comprehension notation $\{x \mid P(x)\}$ to define the input array M. Function isMain: MCC $\longrightarrow$ {true, false} returns true if an MCC is the main module and returns false if otherwise. The notation $C(a_1 = v_1, \ldots, a_m = v_m)$ denotes the instantion of an object of C ($a_i = v_i$ denotes an attribute value assignment). Similarly, assigning (a.k.a attaching [12]) an annotation $A(p_1, \ldots, p_n)$[5] to a model element e amounts to instantiating A by assigning some value v_i to each of the p_i ($i \in [1 \ldots n]$). We write this assignment as: $A(e) : p_1 = v_1, \ldots, p_n = v_n$.

Alg. 1: SCCGen

```
input  : $M = [m_i$ : MCC $\mid i = 1 \ldots n, \exists k \in [1, n]$.isMain$(m_k)]$,
         $N$ : String,  // software name
         $L$ : String   // id label
output : $s$ : SCC s.t SystemDesc($s$).modules = $M$
/* create class $s$ */
1 $s \Leftarrow$ Class(modifier = "public", name = "SCC" + $L$)
2 begin /* assign SystemDesc to $s$ with suitable config */
3   SystemDesc($s$) : appName = $N$,
4     dsDesc = DSDesc(type = "postgresql",
        dsUrl = "//localhost:5432/" + $N$, user = "user",
        password = "password", dsmType = DSM.class,
        domType = DOM.class, osmType = PostgreSQLOSM.class,
        connType = ConnectionType.client),
5     orgDesc = OrgDesc(name = $N$, address = "",
        logo = $N$ + "logo.png", url = ""),
6     modules = $M$
7 return $s$
```

In essense, Alg. 1 consists of two steps: (line 1) create a Class object for s and (lines 2-6) assign SystemDesc to this class to capture the software configuration and, thus, making it an SCC. In particular, line 6 shows that this SCC contains the input MCCs M.

Note that the output SCC s contains the default configuration values for all the properties. Some of these values are derived directly from the input (M, N and L), while others are derived from a combination of these input and pre-defined default values that are supported by the target platform. For example, the data source type (property: DSDesc.type) and the data source's URL (DSDesc.dsUrl)

[5] $p_1, \ldots, p_n$ are annotation properties [12].

are derived from the corresponding JDBC's property values for the PostgreSQL RDBMS. After generation, the configuration values can be customised by the designer to suit the need of the application domain and the platform. In particular, the data source configuration should be customised with the correct user and password. Further, the organisation configuration (`SystemDesc.orgDesc`) would be changed to reflect the actual organisation settings for the software.

4.3 Software Generator

Alg. 2: `SwGen`

```
   input  : s : SCC
   output : w : Software
1  $D \Leftarrow$ initialise DODM from SystemDesc(s).dsDesc;
2  Create the configuration model and its data source (using $D$) ;
3  Use $D$ to create the domain model's data source (containing domain classes
     of all $m_c \in$ SystemDesc($s$).modules) ;
   /* Create the modules and software (w) */
4  foreach $m_c \in$ SystemDesc($s$).modules do
5      create $v$ :ViewConfig, $d$ : ModelConfig, $c$ :ControllerConfig,
         $t$ :ContainmentTree from $m_c$ ;
6      $\mu_c \Leftarrow$ create ModuleConfig composing of $v$, $d$, $c$, $t$ ;
7      create module $m \in$ domMods($w$) from $\mu_c$ ;
8      if isMain($\mu_c$) then $w \Leftarrow m$; /* main module is the software */
9  Show $w$.view ;
10 return $w$
```

We define a jDomainApp's function, named `SwGen`, that automatically generates a software (w) from its SCC (s). The following definition formalises the function's behaviour. Alg. 2 presents a high-level algorithm of the function.

$$\texttt{SwGen} : \texttt{SCC} \longrightarrow \texttt{Software}$$
$$s \mapsto w,$$

with $|w.\texttt{modules}| = |s.\texttt{modules}|$ and
$w.\texttt{dodm} = \texttt{DODM}(\texttt{cfg} = \texttt{SystemDesc}(s).\texttt{dsDesc})$ and
$\texttt{infMod}(w) = \texttt{ModGen}(\texttt{MCCGen}(\texttt{SystemDesc}(s).\texttt{orgDesc}))$ and
$\forall m \in \texttt{domMods}(w), \exists! m_c \in s.\texttt{modules} \bullet m = \texttt{ModGen}(m_c)$.

This definition means that `SwGen` uses the configuration parts captured in `SystemDesc.dsDesc` and `SystemDesc.orgDesc` to initialise the DODM and to create the information module of w (*resp.*). In addition, it uses all the MCCs (m_c) of s to generate the modules (m) of w, such that each m is generated from exactly one m_c. Note that function `infMod: Software` ⟶ `Module` gives the information module of a software. Function `MCCGen: OrgDesc` ⟶ `MCC` overloads the function `MCCGen` that we discussed in Section 3 to automatically generate a suitable MCC from the information contained in an `OrgDesc`. This MCC is used to generate the information module of the software. Function `domMods: Software` ⟶ $\mathcal{P}$(`Module`) gives the set of domain modules of a software.

5 DEVENV INTEGRATION

The objective of DevEnv integration is to make it easier for developers to use a target DevEnv to develop client software with jDomainApp. To prepare for different DevEnvs, we design the jDomainApp's components (displayed in Fig. 2) with a generic interface that can be called upon by the integration component. In this paper, we demonstrate our design with the Eclipse DevEnv[6] and the integration component being in the form of an Eclipse plug-in[7]. In principle, the plug-in consists of a set of handlers for a specific set of events that occur when a developer operates on the source code of the client software. Table 1 summarises the events, handlers and the corresponding jDomainApp's components that are invoked by the handlers. The jDomainApp's components belong to these three module packages: `dcsl`, `mccl` and `sccl`.

Table 1: Eclipse plugin specification

Events	Handlers	Invoked Components
Tool-bar-button clicked	Validate BSpace	`BSpaceValidator`
Source pop-up menu item 1 clicked	Generate BSpace	`BSpaceGen`
Source pop-up menu item 2 clicked	Generate MCC	`MCCGen`
Source pop-up menu item 3 clicked	Generate SCC	`SCCGen`
Source pop-up menu item 4 clicked	Generate software	`SwGen`

5.1 Validating the Behaviour Space

Function `BSpaceValidator: DomainClass` ⟶ `ValidationReport` helps preserve the design integrity of every domain class throughout the development process. Unlike function `BSpaceGen` which is invoked only once to generate the behaviour space, function `BSpaceValidator` is to be invoked at anytime thereafter to check the validity of the behaviour space (*w.r.t* state space), especially when changes to one or both spaces have occured. The function is invoked by clicking the button ✓ on the Eclipse's tool bar. The output is a `ValidationReport`, which contains any errors concerning violations of the domain class design rules. More specifically, an error message details the missing class member(s) (fields and/or methods) that appear in the target of a structural mapping rule between the state and behaviour spaces. These rules are the same as those used by the function `BSpaceGen` to generate the behaviour space. Please refer to [12] for definitions of the rules.

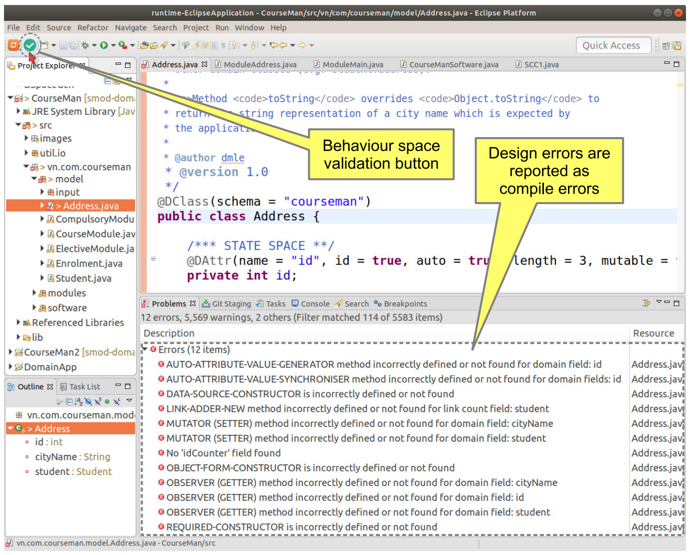
To enforce the rules, we treat the validation errors in the same category as compilation errors and display them in the "Errors" node of the "Problems" tab of the Eclipse workspace. Developers will need to resolve the errors before the class can safely be used. To illustrate, Fig. 4(A) shows how function `BSpaceValidator` is executed against the domain class `Address` of CourseMan. At this stage, this class only has the state space and so the function reports 12 errors on the "Problems" tab. For example, the first error messages states "*AUTO-ATTRIBUTE-VALUE-GENERATOR method incorrectly defined or not found for domain field: id*", which in this case means that the domain method typed "AUTO-ATTRIBUTE-VALUE-GENERATOR" has not been defined for the domain field `Address.id`. The method is required because field `id` in `Address`'s state space is declared to have its values auto-generated.

5.2 The Generators

Figure 4(B) shows how the generators are displayed as menu items in the "Source" sub-menu of a class's context menu. To use the generators to construct a software involves four steps. First, the developer selects the menu-item "Generate BSpace" on each domain class to execute the function `BSpaceGen` for it. Second, the developer chooses one or more domain classes on the "Project

[6]http://www.eclipse.org

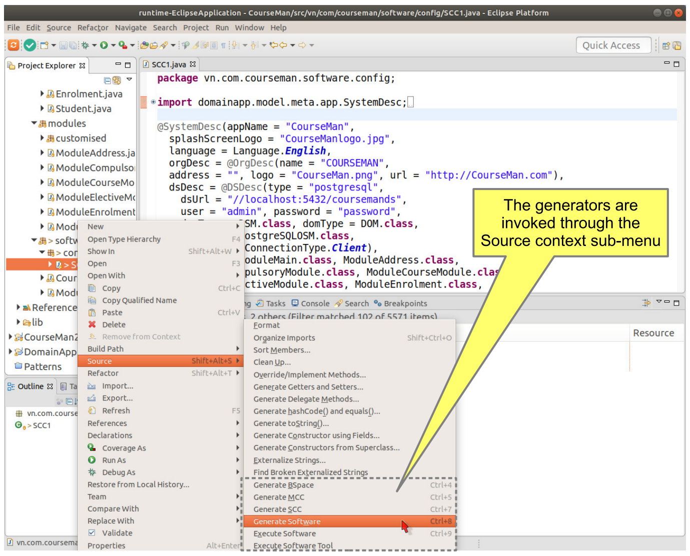
[7]https://www.eclipse.org/pde/


runtime-EclipseApplication - CourseMan/src/vn/com/courseman/model/Address.java - Eclipse Platform
Behaviour space validation button
Design errors are reported as compile errors
Address.java
ModuleAddress.java
ModuleMain.java
CourseManSoftware.java
SCC1.java
Problems
Git Staging
Tasks
Console
Search
Breakpoints

```
 *
 * Method <code>toString</code> overrides <code>Object.toString</code> to
 * return a string representation of a city name which is expected by
 * the applica
 *
 * @author dmle
 * @version 1.0
 */
@DClass(schema = "courseman")
public class Address {

    /*** STATE SPACE **/
    @DAttr(name = "id", id = true, auto = tru  length = 3, mutable =
    private int id;
```

```
12 errors, 5,569 warnings, 2 others (Filter matched 114 of 5583 items)
Description                                                                              Resource
Errors (12 items)
  AUTO-ATTRIBUTE-VALUE-GENERATOR method incorrectly defined or not found for domain field: id      Address.jav
  AUTO-ATTRIBUTE-VALUE-SYNCHRONISER method incorrectly defined or not found for domain fields: id  Address.jav
  DATA-SOURCE-CONSTRUCTOR is incorrectly defined or not found                                      Address.jav
  LINK-ADDER-NEW method incorrectly defined or not found for link count field: student             Address.jav
  MUTATOR (SETTER) method incorrectly defined or not found for domain field: cityName              Address.jav
  MUTATOR (SETTER) method incorrectly defined or not found for domain field: student               Address.jav
  No 'idCounter' field found                                                                       Address.jav
  OBJECT-FORM-CONSTRUCTOR is incorrectly defined or not found                                      Address.jav
  OBSERVER (GETTER) method incorrectly defined or not found for domain field: cityName             Address.jav
  OBSERVER (GETTER) method incorrectly defined or not found for domain field: id                   Address.jav
  OBSERVER (GETTER) method incorrectly defined or not found for domain field: student              Address.jav
  REQUIRED-CONSTRUCTOR is incorrectly defined or not found                                         Address.jav
```

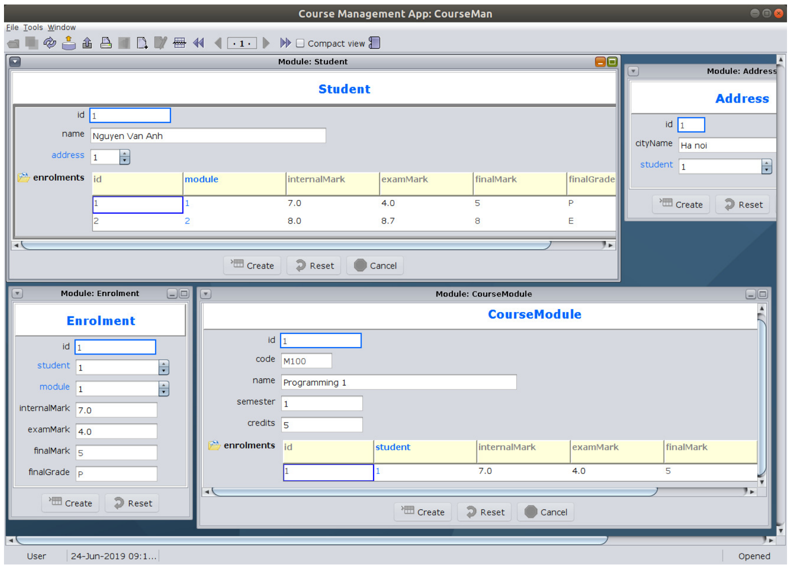

runtime-EclipseApplication - CourseMan/src/vn/com/courseman/software/config/SCC1.java - Eclipse Platform
The generators are invoked through the Source context sub-menu
Format
Organize Imports
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate toString()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Externalize Strings...
Find Broken Externalized Strings
Generate BSpace
Generate MCC
Generate SCC
Generate Software
Execute Software
Execute Software Tool

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

import domainapp.model.meta.app.SystemDesc;

@SystemDesc(appName = "CourseMan",
  splashScreenLogo = "CourseManlogo.jpg",
  language = Language.English,
  orgDesc = @OrgDesc(name = "COURSEMAN",
  address = "", logo = "CourseMan.png", url = "http://CourseMan.com"),
  dsDesc = @DSDesc(type = "postgresql",
    dsUrl = "//localhost:5432/coursemands",
    user = "admin", password = "password",
          SM.class, domType = DOM.class,
          ostgreSQLOSM.class,
          ConnectionType.Client),
          oduleMain.class, ModuleAddress.class,
          pulsoryModule.class, ModuleCourseModule.clas
          ctiveModule.class, ModuleEnrolment.class,
```

Figure 4: The Eclipse plugin GUI: (A-Left) Using `BSpaceValidator` **to validate a domain class; (B-Right) Invoking the generators through the source context sub-menu.**



id	name	address	
1	Nguyen Van Anh	1	

id	module	internalMark	examMark	finalMark	finalGrade
1	1	7.0	4.0	5	P
2	2	8.0	8.7	8	E

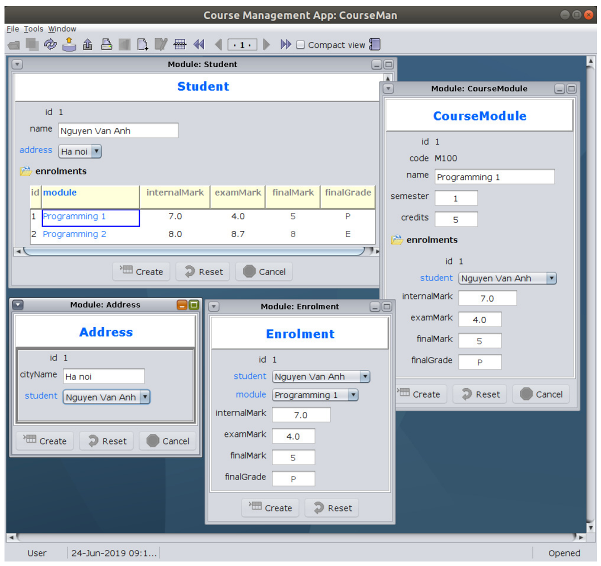
id	cityName	student
1	Ha noi	1

id	student	module	internalMark	examMark	finalMark	finalGrade
1	1	1	7.0	4.0	5	P

id	code	name	semester	credits
1	M100	Programming 1	1	5

id	student	internalMark	examMark	finalMark
1	1	7.0	4.0	5

User 24-Jun-2019 09:1... Opened



id	name	address
1	Nguyen Van Anh	Ha noi

id	module	internalMark	examMark	finalMark	finalGrade
1	Programming 1	7.0	4.0	5	P
2	Programming 2	8.0	8.7	8	E

id	code	name	semester	credits
1	M100	Programming 1	1	5

id	student	internalMark	examMark	finalMark	finalGrade
1	Nguyen Van Anh	7.0	4.0	5	P

id	cityName	student
1	Ha noi	Nguyen Van Anh

id	student	module	internalMark	examMark	finalMark	finalGrade
1	Nguyen Van Anh	Programming 1	7.0	4.0	5	P

User 24-Jun-2019 09:1... Opened

Figure 5: Two CourseMan software's GUIs: (A-Left) Default configuration, (B-Right) Customised configuration

Explorer" and select the menu-item "Generate MCC" to generate the MCCs from these classes. Third, the developer chooses one or more MCCs on the "Project Explorer" and select the menu-item "Generate SCC" to generate an SCC for a software. Fourth, the develper chooses an SCC and select the menu-item "Generate software" to generate and execute a software. Figure 5(A) shows the CourseMan software's GUI that is generated automatically using the configurations in the automatically-generated MCCs. Figure 5(B), on the other hand, shows the same software but with its GUI has been significantly improved by customising the view configurations in the MCCs. In particular, the view layouts of `ModuleStudent` and `ModuleCourseModule` have been made more compact using a different layout builder class. All the reference-typed view fields (including `Student.address`, `Address.student` and `Enrolment.student`) have been changed to combo-style fields that display name values of the referenced objects. Further, the sub-view `CourseModule.enrolments` has been changed from a tabular component to a 2-column panel component. Note that all the view field labels can also easily be customised to contain more user-friendly texts.

6 EVALUATION

We focus our evaluation on a new key contribution of this paper, which is software generation (package: `sccl`). The other two core packages (`dcsl` and `mccl`) have been evaluated in previous works [12, 13]. Among the two functions in `sccl`, `SCCGen` can trivially be judged to have a constant time complexity. Thus, we will focus on evaluating function `SwGen`.

Method. Although it can be observed without much difficulty that the two dominating steps (steps 3 and 4) of Alg. 2 have linear time complexities in the module set size, we choose experimentation over theoretical analysis of `SwGen`. A main reason is because we want to know the actual time it takes to generate the software. Another reason is because there are steps in the algorithm that depend on third-party libraries and external systems, whose performances affect `SwGen` but are outside of its control. For example, steps 1 and 3 depend on the performance of the JDBC library and the PostgreSQL RDBMS.

We measure `SwGen`'s performance using a set of test software. Each software is a variant of CourseMan, whose domain model contains an increasing number of copies of the CourseMan domain model (see Figure 1). We use the Eclipse plugin to conveniently create for each variant a domain model, the MCCs from this model and the SCC from these MCCs. We then execute `SwGen` on each SCC to generate and run a software. We measure the total execution time (in seconds) and allocated memory space (in megabytes) for each variant and plot them on two separate charts. We conduct our experiment on a Dell Vostro laptop, running the Ubuntu[8] 18.04.2 (Bionic) operating system. The Java virtual machine version compatible with jDomainApp is 8.0[9]. We use the Google's Caliper [10] micro-benchmarking tool for Java to measure `SwGen`. The hardware configuration includes 8GB RAM and a 64-bit Intel CPU (core i7, 8th generation), running at 1.80GHz.

Note that we do not directly compare our `SwGen`'s performance to other DDD frameworks [3, 18], because, unlike `SwGen`, these frameworks do not generate software from modules.

Data set. The data set consists of 14 software variants listed in the first column of Table 2. To ease discussion, we label each variant by the number of domain classes contained in its domain model. For example, variant 6 has 6 classes, variant 300 has 300 classes (i.e. 50 copies of the basic model), and so on. We divide the dataset into three (overlapping) subsets, two of which are evenly paced, but with different increment values. The first subset contains the first 10 software variants (6–60), which are evenly-paced with the increment of 6 (classes or 1 copy). The next two subsets are designed to test the rate of change, using much larger increments. The second subset contains the next three variants (60, 120 and 300), whose increments are to approximately double the model size. The third subset contains the last three variants (300–900), which are evenly-paced with the increment of 300 (classes). We stopped the experiment at variant 900 because, beyond this, the memory requirement exceeds (4.5GB) the available memory of the test computer.

Table 2: Data set and results

Software variants	Time (secs)	Memory (MBytes)
6	0.928	14.62
12	0.935	19.15
18	0.975	24.33
24	0.977	30.61
30	0.994	35.80
36	1.047	42.35
42	0.982	48.62
48	1.053	55.28
54	1.089	62.24
60	1.083	69.69
120	1.100	162.17
300	1.401	661.03
600	3.254	2,146.28
900	5.332	4,408.46

Result. The execution times and memory allocations of each software variant are recorded in the second and third columns of Table 2 (*resp.*). Figures 6a and 6b show the two charts that depict these results. We observe that the overall shapes of both charts appear similar and can be approximated by a linear trend line. This stems from two facts. First, the two main chart segments that correspond to the two evenly-paced subsets of the data set are very close to a linear shape. Second, the chart segment depicting the remaining (second) subset approximately fills the gap between the other two segments.

In brief, we are confident that SwGen has linear-time complexity and is, thus, scalable to handle large real-world software.

7 RELATED WORK

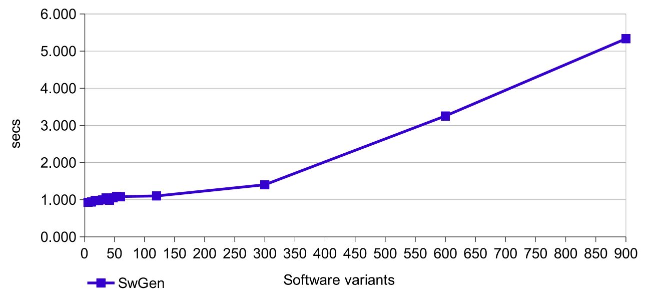
Our work in this paper is positioned at the intersections among the three areas: GSD, DSL engineering and DDD and frameworks.

GSD. Of particular interest to our paper are GSD [2] approaches that use DSL to express software configuration. In light of the recent GSD development [1], we would argue that a software in our method is similar to a product in product-line engineering. A software module is analogous to a feature, which is defined generally as "...a characteristic or end-user-visible behavior of a software system" [1]. Further, our proposed sccl language is an aDSL-derived feature modelling language: an SCC is equivalent to a feature model. Our proposal is unique in its weaving of aDSL and traditional module-based object-oriented software development. Our method helps significantly reduce development effort by leveraging the host OOPL's capabilities (language and tools).

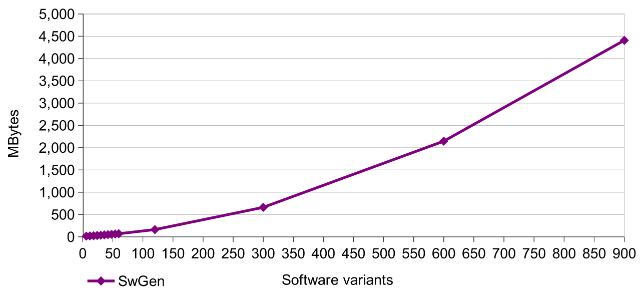
DSL Engineering. Our work concerns with DSL engineering [9] approaches that apply meta-modeling [8] to define an aDSL. Nosal et al. [15] discuss three general approaches for developing aDSL: (*i*) language unification (annotation is used to define the domain concepts) (*ii*) language referencing by extension (annotation defines references to the concepts in another language) (*iii*) language extension (annotation defines new concepts that do not exist independently from the host language). Our development method for sccl is actually a combination of the first and second approaches. Further, we applied meta-modelling with UML to specify the sccl's abstract syntax. Nosal et al. [15] do not address how meta-modelling is used to define aDSL.

DDD and frameworks. We argue that our proposed method and techniques in this paper help make the original DDD method more concrete and more complete. On the one hand, the DDD method [4, 5] and its recent developments [14, 21] mainly focus on domain modelling and do not address software construction from the domain model. Further, the DDD method does not provide any

[8]http://www.ubuntu.com
[9]http://java.oracle.com
[10]http://github.com/google/caliper



(a) Execution time.



(b) Memory space allocation.

Figure 6: The experiment results for `SwGen`.

specific DSLs for domain modelling. On the other hand, to the best of our knowledge, recent DDD frameworks [3, 18] do not address software construction generatively and from the module-based perspective. In particular, they do not treat modules as first-class objects and lack a method for the construction.

8 CONCLUSION

In this paper, we proposed a novel solution to tackle the problem of generative, module-based software construction in DDD and the integration of this in a DevEnv. Our solution consists of a software framework, named jDomainApp, and an Eclipse plugin that integrates the jDomainApp's software construction functions into the Eclipse DevEnv. jDomainApp is built on a module-based MVC architecture. A module is an MVC component, which is instantiated from a module class. A software consists of a set of modules. To enable automatic software construction using the domain model at the core, reusing the available jDomainApp's components, we treated software as an instance of a software class (a software product family) and applied the generative software development approach. Our method is novel in the use of three aDSLs: DCSL (for expressing the domain model), MCCL (for generating module class) and sccl (for generating software class). The first two languages were developed in our previous works, the last language was developed in this paper. Together, these aDSLs extend the jDomainApp's core functionality and provide a basis for our design of the Eclipse plugin. The plugin enables a developer to automatically and incrementally perform all the essential software construction functions directly from the domain model: generating and validating the domain classes, creating MCCs from the domain classes, creating an SCC from a set of MCCs and creating and executing a software from its SCC. We evaluated the software construction's performance to show that it is scalable to handle large real-world software.

We contend that our work in this paper constitutes a novel and significant contribution to the DDD method, which helps makes the method more concrete and more complete. We have been using jDomainApp in developing real-world software and in teaching software engineering course modules at the Faculty of IT, Hanoi University. We plan to take our work to the industry and seek opportunities to apply it to develop industrial-scale software.

REFERENCES

[1] Sven Apel, et al. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.
[2] Krzysztof Czarnecki. 2005. Overview of Generative Software Development. In *Unconventional Programming Paradigms.* Number 3566 in LNCS. Springer, 326–341.
[3] Dan Haywood. 2013. Apache Isis - Developing Domain-driven Java Apps. *Methods & Tools: Practical knowledge source for software development professionals* 21, 2 (2013), 40–59.
[4] Eric Evans. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.
[5] Eric Evans. 2014. *Domain-Driven Design Reference: Definitions and Pattern Summaries.* Dog Ear Publishing, LLC.
[6] James Gosling, et al. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional, Upper Saddle River, NJ.
[7] Anders Hejlsberg, et al. 2010. *The C# Programming Language* (4th ed.). Addison Wesley, Upper Saddle River, NJ.
[8] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels* (1st ed.). Addison-Wesley Professional, Upper Saddle River, NJ.
[9] Tomaž Kosar, et al. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (2016), 77–91.
[10] Glenn E. Krasner et al. 1988. 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.
[11] Duc Minh Le, et al. 2017. Generative Software Module Development: A Domain-Driven Design Perspective. In *Proc. 9th Int. Conf. on Knowledge and Systems Engineering (KSE).* 77–82.
[12] Duc Minh Le, et al. 2018. On Domain Driven Design Using Annotation-Based Domain Specific Language. *Computer Languages, Systems & Structures* 54 (2018), 199–235.
[13] Duc Minh Le, et al. 2019. Generative Software Module Development for Domain-Driven Design with Annotation-Based Domain Specific Language. *(Conditionally Accepted) Journal of Information and Software Technology* (2019).
[14] Scott Millett et al. 2015. *Patterns, Principles, and Practices of Domain-Driven Design.* John Wiley & Sons.
[15] Milan Nosál', et al. 2016. Language Composition Using Source Code Annotations. *Computer Science and Information Systems* 13, 3 (2016), 707–729.
[16] OMG. 2015. Unified Modeling Language version 2.5. http://www.omg.org/spec/UML/2.5/
[17] Oracle. 2018. Java Platform, Standard Edition Java Language Updates.
[18] Javier Paniza. 2011. *Learn OpenXava by Example.* CreateSpace, Paramount, CA.
[19] Dirk Riehle et al. 1998. Role Model Based Framework Design and Integration. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 117–133.
[20] Ian Sommerville. 2011. *Software Engineering* (9th ed.). Pearson.
[21] Vaughn Vernon. 2013. *Implementing Domain-Driven Design* (1st ed.). Addison-Wesley Professional, Upper Saddle River, NJ.