Towards Generating RESTful Full Stack Software: A Module Based Domain Driven Design Approach

Duc Minh Le[1] and Ha Thanh Vu[2]

[1] Department of Software Engineering, Swinburne Vietnam, FPT University
`duclm20@fe.edu.vn`
[2] Design and Consulting Centre, MobiFone, Hanoi, Vietnam
`ha.vuthanh@mobifone.vn`

Abstract. Domain-driven design (DDD) has widely been used to develop RESTful software in a range of programming language platforms. The use of code generators in these technologies helps significantly increase productivity and achieve large scale software reuse. However, there been no work that address incremental construction of RESTful full stack software (RFS) at both the module and software levels. In this paper, we propose a generative module-based method for RFS to bridge this gap. We characterise RFS and realise this in a module-based DDD software architecture, named MOSA^R. Our method takes as input a software configuration, written in an annnotation-based domain specific language, and automatically generates a module-based RFS software. We present algorithms for the frontend and backend generation functions. The backend software consist of web service modules. The frontend software consists of modules that are designed with single-page views. Each view can be nested to reflect the containment tree of the module. We demonstrate method by implementing the generators for two popular platforms: React (frontend) and Spring Boot (backend). We evaluate the generators' performance to show that they are scalable to support large software.

Keywords: domain-driven design, software framework, system modelling, system configuration.

1 Introduction

It is without questions that rapid evolution of modern object-oriented programming languages (OOPLs), such as Java and C#, has made domain-driven design (DDD) [5,22,12] a dominant force in the industry over the past two decades. The basic DDD's tenet is clear and simple: the core (a.k.a "heart") of software is the domain model and, thus, effectively constructing this model is a central issue. The addition of the *annotation*[3] construct in OOPL brings new modelling capabilities [15,9] that help ease the development of domain model and software in this type of language.

[3] What Java calls *annotation* is *attribute* in C#.

DDD has widely been used to develop web-service-based software in OOPLs (e.g. C# [23] and Java [13]). In particular, a number of software frameworks, most noticably ApacheIsIs [3], OpenXava [7] and jHipster [11], have been developed to make RESTful [6] software development in DDD more productive. The use of code generators in these frameworks helps significantly increase productivity and achieve large scale software reuse. However, there been very few methods that focus on RESTful full stack software (RFS) and none of these methods support incremental software generation at both the module and software levels. In this paper, we propose a generative module-based method to construct RFS to bridge this gap. We first characterise RFS and realise this in a software architecture, named **MOSA**^{R}, which we extend from our previously-developed module-based MOSA architecture for DDD. Our method takes as input a software configuration, written in an annnotation-based domain specific language, and automatically **generates** a module-based RFS software. The backend software consist of web service modules. The frontend software consists of modules that are designed with single-page views. Each view can be nested to support the containment tree of the module. We presents algorithms for two generator functions that generate the front- and back end software. To demonstrate our method, we implement the generators for two popular programming platforms: React (frontend) and Spring Boot (backend). We evaluate the generators' performance to show that they are scalable to support large software.

The rest of the paper is structured as follows. Section 2 explains a motivating example and presents some key background concepts about MOSA. Section 3 discusses the characteristics of RFS and presents MOSA^R. Section 4 discusses our RFS generator functions. Section 5 presents the evaluation result. Section 6 reviews the related work and Section 7 concludes the paper.

2 Background on MOSA Architecture

In this section, we present a motivating software example named CourseMan and a number of key concepts about the MOSA architecture.

2.1 Motivating Example: CourseMan

To illustrate the concepts presented in this paper, we adopt the software example, named course management (CourseMan), that we used in previous works [8,10,9]. This example is scoped around a domain model whose elements are represented by the following fundamental UML [16] meta-concepts: class, attribute, operation, association, association class and generalisation.

Figure 1 shows the *domain model* of CourseMan. The domain model consists of six domain classes and associations between them. For brevity, we exclude operations from the class boxes. Class `Student` represents the domain concept Student[4]. Class `CourseModule` represents the `CourseModules`[5] that are offered

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

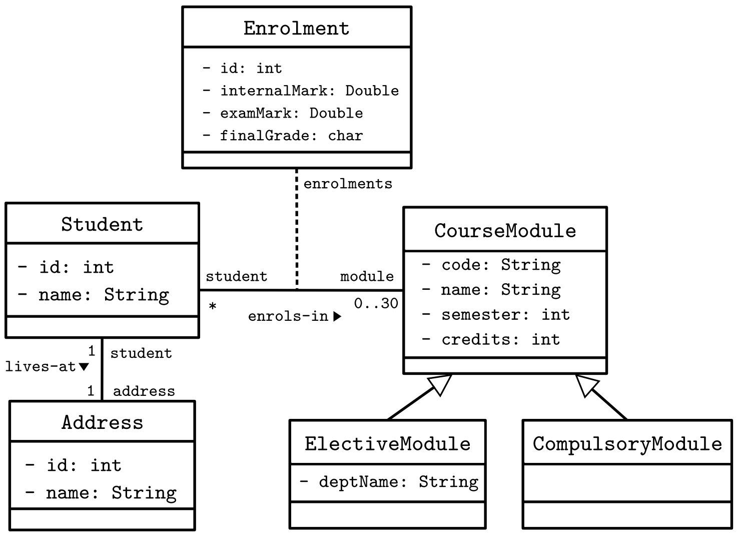


Fig. 1: The core CourseMan domain model.

to students. Class `Address` represents the addresses where `Student`s live as they undertake their studies. The study arrangement involves enrolling each `Student` into one or more `CourseModule`s. This many-many association is resolved by an association class named `Enrolment`, which contains additional information about the aggregate marks and the final grade.

2.2 MOSA Architecture Model

Figure 2 is a UML class model that represents the abstract view of the module-based software architecture (MOSA) [8,9]. MOSA is an MVC software architecture for DDD, which consists of three layers: domain model (the core), module and software. A **domain model** essentially consists of a set of domain classes and the associations between them. We define this model by an aDSL named DCSL. This language expresses the design space of a domain class in terms of *state*

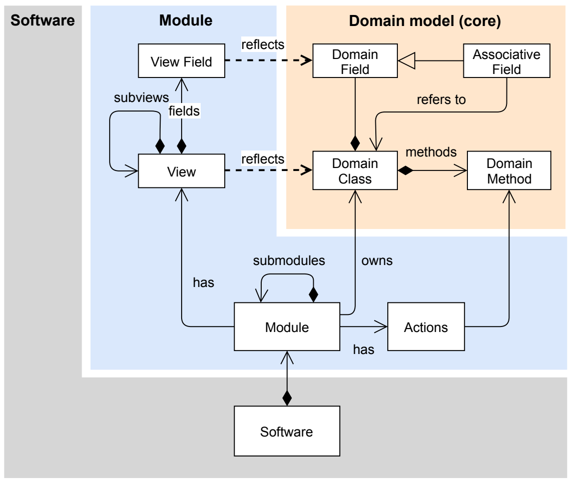


Fig. 2: The abstract MOSA architecture.

space and *behaviour space*. The state space, in particular, consists of all the domain fields. A software is incrementally constructed from its modules. A module is a ‘little’ MVC-based software, whose model is a domain class. Modules are instantiated from a *module class*. A module’s structure is constructed from a *rooted containment tree*, whose non-root nodes are *descendant modules* that own the domain classes associated (directly or indirectly in an *association chain*) to the module’s owned domain class.

We adapted the *software product-line development* (SPD) approach [4] to generatively construct software and its modules from their configurations. We use *annotation-based domain-specific languages* (aDSLs) [15] to expess these configurations. To generate the module class we defined an aDSL, named MCCL, to express *module class configuration* (MCC). The MCC is automatically generated from a domain class and then customised to satisfy the required module design specification. In this paper, we use the term MCC synonymously to module class. To generate a software class from the modules we defined another aDSL, named SCCL, to express the *software configuration classes* (SCCs). Among the essential properties of an SCC is `modules`, which specifies the set of MCCs of the modules that make up the software class.

For example, the aforementioned CourseMan domain model results in six MCCs, each of which is owns one domain class. The view of `ModuleStudent`, for instance, has 2 view fields that represent the two domain fields: `Student.id` and `name`. In addition, this view contains two subviews of the following level-1 nodes of `ModuleStudent`’s containment tree: `ModuleAddress` and `ModuleEnrolment`.

3 MOSA^R: Extending MOSA for RFS Software

Given that REST-style architecture [6] is the underpining architecture of the modern Web. A **RESTful full stack** (**RFS**) **software** is a web-based software whose frontend (FE) and backend (BE) fully comply with a REST-style architecture. Figure 3 presents our proposed MOSA^R architectural model, which extends MOSA for RFS. MOSA^R uses the same core domain model layer as MOSA. However, the module and software layers are restructured to support frontend, backend and RESTful API.

3.1 Characteristics of RFS

Featurewise, in this paper we will focus on the following component categories of FE and BE that were reported by Taivalsaari et al. [20]:

1. FE components: web frontend components that are constructed in HTML/CSS/JavaScript and using at least one popular frontend frameworks (including Angular, React and Vue)
2. BE components: database-enabled domain-logic-processing components that are constructed in at least one popular backend frameworks
3. RESTful API components: to enable REST-style communication between FE and BE

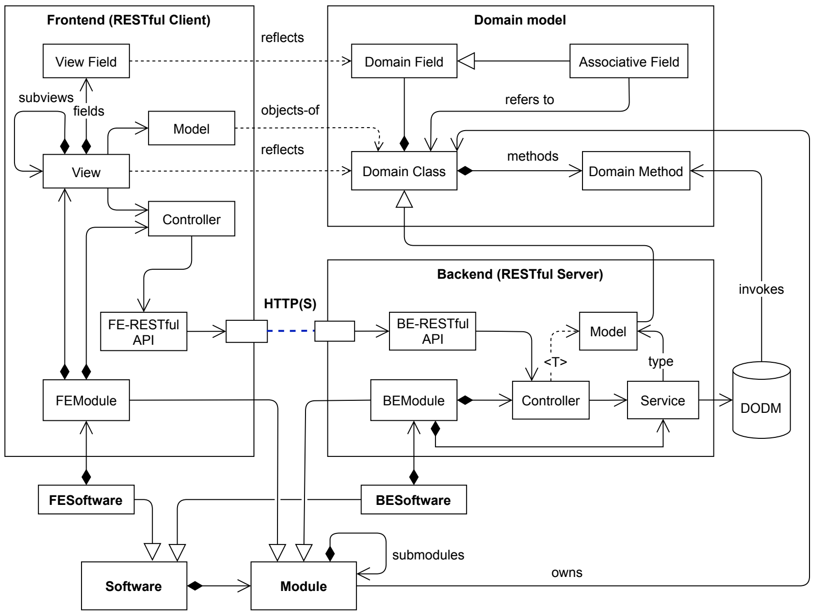


Fig. 3: The MOSA^R architecture model.

4. Independent platforms: the FE and BE components can be constructed and deployed to different language platforms

To ease discussion, in the remainder of this paper we will refer to the technologies used by the frontend components (including HTML/CSS/JavaScript and frontend frameworks) as **frontend technologies**. Similarly, we will refer to the technologies used by the backend components as **backend technologies**.

3.2 FrontEnd Software

`FESoftware` is a subtype of `Software` that consists in a set of FE modules. Conceptually, an `FEModule` is a subtype of `Module` that uses frontend technologies to construct the `View`, `Model` and `Controller` components. In modern frontend frameworks, `FEModule` consists of a `Model` component, which is a *mirror representation* of the domain class's state space. That is it consists in the same core structure as the state space, but is expressed in terms of a frontend language. In MOSA^R, `FEModule`'s `Model` captures part of the `View`'s state that pertains to the domain objects manipulated by the user on the view. Conceptually, `Controller` is a cohesive set of event-handling methods that are invoked when certain events concerning `Model` are fired. As shown in Fig. 3, `Controller` uses a frontend RESTful API to connect to the backend.

Single Page View

In this work, we focus on a modern design of the `FEModule`'s `View`, named **single page view** (**SPV**) [19]. In this design, all the necessary `View` components are constructed before hand and downloaded into the client's web browser. Subsequent changes to the view concerning the view state are handled as part of the communication with the backend.

Conceptually, a `View` *w.r.t* an MCC m is the tuple $\langle c', V_I, \mathcal{V}_C \rangle$, where c' is the front-end representation of m's domain class. V_I is the **index** or 'home' component that is presented to the user when they first access `FEModule`. This component's content is constructed from other components specified in the set $\mathcal{V}_C$. Each component in this set is responsible for providing one view aspect. Within the scope of this work, we consider two basic view aspects: form (V_F) and browser (V_B). V_F is the **form** component that presents an input form for the user to create a new domain object or to edit an existing one. V_B is the **browser** component that basically presents a table of the existing domain objects for the user to browse. Both V_F and V_B allow the user to select an object to view its details on V_F. It also allows the user to delete objects. We will discuss in Section 4.1 an algorithm for generating this type of view.

3.3 BackEnd Software

`BESoftware` is a subtype of `Module` that consists in a set of BE modules. An **BEModule** is a subtype of `Module` that uses backend technologies to handle the domain-specific logic in response to `FEModule`'s request. In the context of DDD and REST, `BEModule`'s functionality is defined by a set of web services. Unlike `FEModule`, however, `BEModule` typically does not include the `View`.

We thus define a `BEModule` *w.r.t* an MCC m as the tuple $\langle c, \mathcal{C}, \mathcal{S} \rangle$, where $c =$ m's domain class, $\mathcal{C}$ is the set of `Controllers` of the modules in m's containment tree and $\mathcal{S}$ is the set of `Services` that constitute m's functionality. `Controller` basically acts as a router that typically routes requests to the relevant `Services` to handle. This pattern also applies to `Controllers` of the descendant modules that participate in the containment tree of a `BEModule`. Similar to MOSA, both `Controller` and `Service` are parameterised with the domain class of the module, so that they can handle the domain objects of this class. In MOSA^R, the set $\mathcal{S}$ contains at least the repository service [5], which uses the MOSA's DODM component to manage objects and their storage.

3.4 RFS Configuration

We extend SCC to define SCC^R for MOSA^R. SCC^R includes an annotation, named `RFSDesc`, which specifies the configuration for RFS generation.

Listing 1.1: An illustration of `RFSDesc`'s core structure

```
@RFSDesc(
  domain = "courseman",   // software domain name
  feLangPlatform = LangPlatform.REACT,  // target frontend language platform
  beLangPlatform = LangPlatform.SPRING, // target backend language platform
  outputPath = "src/example/java" // output path for the generated components
)
```

In this work, we define `RFSDesc` in terms of 4 basic properties. Other properties would be added in the future to accommodate the configuration of new MOSA^R's features. Listing 1.1 gives an example that illustrates `RFSDesc`. The embedded comments provide descriptions of the properties.

4 RFS Generation

In this section, we discuss an adaptation of our template-based software generation method [10] for RFS. A unique feature of our method is to use the same core domain model as the input and to incrementally and automatically generate the `FESoftware` and `BESoftware` from their modules. We denote by $\texttt{SCC}^R$ the set of all the SCC^{R}s concerning a software domain. Further, we denote by $\mathcal{A}$ a Cartesian product of sets of domain and software assets that are referenced (directly or indirectly) by an SCC^R. Among the **required assets** include the domain model and the FE and BE component templates.

4.1 Generating FrontEnd Software

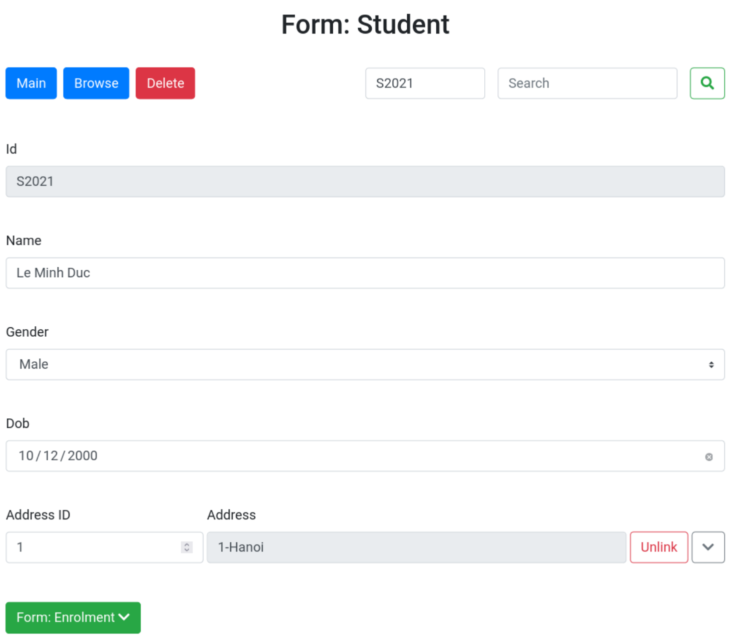
We adopt the SPV design for FE (see Section 3.2) and define the FE generation function: `FEGen`: $\texttt{SCC}^R \times \mathcal{A} \rightarrow$ `FESoftware`. This takes as input an SCC^R with the required assets $A \in \mathcal{A}$ and generates as the output an `FESoftware` consisting of `FEModule`s, whose views are designed with SPV.

Alg. 1: `FEGen`: `FESoftware` as a set of single-page `FEModules`

```
   input  : $s : \texttt{SCC}^R, A \in \mathcal{A}$
   output: $W_F$ : FESoftware
1  $W_F \leftarrow \emptyset$                                  /* the output FESoftware */
2  foreach $m_c \in \texttt{mccs}(s)$ do
3      $V_F \leftarrow \texttt{genViewForm}(m_c, A)$                   /* Form view */
4      $V_B \leftarrow \texttt{genViewBrowser}(m_c, A)$             /* Browser view */
5      $V_I \leftarrow \texttt{genViewIndex}(V_F, V_B, A)$            /* Index view */
6      $W_F \uplus \{\langle V_I, V_F, V_B \rangle\}$
7  return $W_F$
8  Function genViewForm($m_c$ : MCC, $V_F$ : ViewForm, $A : \mathcal{A}$)
9      $V_F \leftarrow \texttt{loadTemplate}(\text{“view\_form”}, A)$ /* init form template */
10     $P \leftarrow \emptyset$                               /* view components set */
11     $c \leftarrow m_c.\texttt{domainClass}$
12     foreach $f \in c.\texttt{fields}$ do
13         $f_d \leftarrow \texttt{fieldDef}(m_c, f)$
14         if $\texttt{isNonAssoc}(f)$ then /* generate view field */
15             $v \leftarrow \texttt{genViewField}(f, f_d, A)$ ; $P \uplus \{v\}$
16         else /* generate subview */
17             $v \leftarrow \texttt{genSubView}(f, f_d, A)$ ; $P \uplus \{v\}$
18     $\texttt{rewrite}(V_F, \text{“title”}, m_c.\texttt{title})$ ; $\texttt{rewrite}(V_F, \text{“view.comps”}, P)$
19     return $V_F$
```

Alg. 1 presents the algorithm that realises this function. The **foreach** loop in the top half of the algorithm creates, for each MCC (m_c), an `FEModule` as a single-page component consisting in three views: form view (V_F), browser view (V_B) and the index view (V_I). The three views are generated at lines 3–5 using pre-defined view templates. Function `genViewBrowser` is a simple adaptation of `genViewForm`, while function `genMainView` basically involves rewriting a `ViewIndex` template using the source codes of the two component views. The pseudocode for `genViewForm` is presented in the bottom half of Alg. 1. It basically implements a template-based code generation logic. After loading the template at line 9, its **foreach** loop iterates over the domain fields of the module's domain class (c) to create a view component (v) for each field. This can be either a normal field (line 15) or a subview of a descendant module (line 17). After all the view components have been created, they are inserted into V_F (by function `rewrite` at line 18), replacing the template variable "`view.comps`". V_F's title is also rewritten (line 18) using title in m_c.

Two properties of Alg. 1 worth highlighting are *recursiveness* and *view reusability*. First, it is recursive *w.r.t* function `genViewForm`. This function is called by the sub-view generation function (line 17) if the containment tree extends to include associations to other domain classes.



S2021

Id: S2021

Name: Le Minh Duc

Gender: Male

Dob: 10/12/2000

Address ID: 1

Address: 1-Hanoi

Fig. 4: The generated form view of `ModuleStudent`.

Example 1. (The form view of `ModuleStudent`).

Figure 4 shows the rendered form view of `ModuleStudent`. The source code of this view, which is generated by function `FEGen` for the React.js platform[6] is given in Listing 1.2. The source code uses the popular CSS-based design named BootStrap[7]. To ease reading, however, we omit from the listing all the CSS-specific

[6] https://reactjs.org
[7] React-Bootstrap: https://react-bootstrap.github.io/

style and event handling elements. The complete source code of the example is given as part of our implementation on GitHub. Line 3 of the listing shows the function, named `renderTitle`, that renders the form title as “`Form : Student`”. Line 4 shows the main function, named `renderForm`, that renders the `Student` form content. Each view field (e.g. `Student.id`) is basically rendered as a (label, form-control) pair (e.g. the pair at lines 7, 8). Each subview (e.g. `Address` at line 17 and `Enrolment` at line 21) is rendered by reusing the view of the corresponding descendant module.

Listing 1.2: The generated source code of `ModuleStudent`’s form view in React.js

```
 1 export default class StudentForm extends BaseForm {
 2   // code omitted
 3   renderTitle() { return (<> Form: Student </>); }
 4   renderForm() {
 5     return (<> <br />
 6     <FormGroup>
 7       <Form.Label>Id</Form.Label>
 8       <FormControl type="text" value={this.renderObject("current.id")}/>
 9     </FormGroup> <br />
10     <FormGroup’>
11       <Col>
12         <Form.Label>Address ID</Form.Label>
13         <FormControl type="number" value={this.renderObject("current.
              addressId")} /></Col>
14       <Col>
15         <Form.Label>Address</Form.Label>
16         <FormControl type="text" value={this.renderObject("current.address")}
               /></Col>
17       <AddressSubmodule compact={true} mode=‘submodule’ title="Form: Address"
18         current={this.props.current.address}
19         parentName=‘student’ parent={this.props.current}/>
20     </FormGroup> <br />
21     <EnrolmentSubmodule mode=’submodule’ title="Form: Enrolment"
22       current={this.props.current.enrolments}
23       parentId={this.props.currentId} /> </>);
24   }
25 }
```

□

4.2 Generating BackEnd Software

We define the BE generation function: `BEGen`: $\mathsf{SCC}^R \times \mathcal{A} \to$ `BESoftware`, which takes as input an SCC^R with the required assets $A \in \mathcal{A}$ and generates a `BESoftware` as the output. Alg. 2 presents the algorithm of this function. The main **foreach** loop creates a `BEModule` from each MCC m_c of the input SCC^R.

Alg. 2: BEGen: BESoftware as a set of BEModules

```
   input  : $s : \mathsf{SCC}^R, A \in \mathcal{A}$
   output: $W_B$ : BESoftware
 1 $W_B \leftarrow \emptyset$                          /* the output BESoftware */
 2 foreach $m_c \in$ mccs($s$) do
 3     $S \leftarrow$ genRepoServiceType($m_c, A$); $\mathcal{S} \leftarrow \{S\}$     /* RESTService */
 4     $C \leftarrow$ genControllerType($m_c, A$) ; $\mathcal{C} \leftarrow \{C\}$     /* RESTController
        */
 5     $T \leftarrow$ containmentTree($m_c$)
 6     if $T \neq \emptyset$ then /*gen. controllers of descendant mods in $T$*/
 7         foreach $m_d \in T \wedge m_d \neq T$.root do
 8             $C_d \leftarrow$ genDescendantControllerType($m_c, m_d, A$)
 9             $\mathcal{C} \uplus \{C_d\}$
10     $c \leftarrow m_c$.domainClass
11     $W_B \uplus \{\langle c, \mathcal{C}, \mathcal{S}\rangle\}$
12 return $W_B$
```

Alg. 2 presents the algorithm that realises this function. The main **foreach** loop creates a `BEModule` from each MCC m_c of the input SCC^R. Line 3 uses the function `genRepoServiceType` to generate a repository service class (using a template available in A) and adds it to a services set ($\mathcal{S}$). Calls to other functions can be added here to generate additional services.

Similarly, line 4 uses the function `genControllerType` to generate the corresponding controller class for m_c and adds it to the controller set ($\mathcal{C}$). This set may also contain the controller classes of the descendant modules (m_d) in the containment tree T of m_c. These controllers (C_d) are generated (if not already) by the function `genDescendantControllerType` in the nested **foreach** loop that starts at line 7.

The three aforementioned generator functions basically involve creating a subtype of a pre-defined service or controller type and overriding the default methods already implemented in the supertypes. For CRUD functionality, this results in full generation. For more complex domain-specific functionality, to achieve full generation requires extending the generator function with add-on components that implement the extra functionality. We implemented this design in our previous work [9] with the MOSA architecture. We plan to adapt it for RFS in future work.

Example 2. (The generated back-end of `ModuleStudent`).

We describe in this example the backend source code of `ModuleStudent`, which is generated by `BEGen` for the SpringBoot platform [17]. Listings 1.3 and 1.5 show the generated code of the controller class and the default service class of `ModuleStudent`. Listing 1.4 shows the generated source code of the controller class of the descendant module `ModuleEnrolment`, that participates in `ModuleStudent`'s containment tree. As can be seen from the listings, the two controller classes are tagged with @`RestController`, which are treated by SpringBoot as RESTful controller components. Similarly, the service class is tagged with @`Service`. The implementation of both controller classes are straight-forward, whose methods simply invoke the provided super-types' methods. The CRUD requests are mapped to methods that are tagged with the corresponding request mapping annotations. Note that both the controller and service classes support a method to handle the pagination request. This request is used by the frontend's browser view to obtain a fixed-sized collection of objects to present to the user.

Listing 1.3: The controller of `ModuleStudent`

```
@RestController()
@RequestMapping(value = "/students")
public class StudentController extends DefaultRestfulController<Student> {
  @GetMapping()
  public Page getEntityListByPage(PagingModel arg0) {
    return super.getEntityListByPage(arg0);
  }
  @PostMapping()
  public Student createEntity( @RequestBody() Student arg0) {
    return super.createEntity(arg0);
  }
  @GetMapping(value = "/{id}")
  public Student getEntityById(Identifier arg0) {
    return super.getEntityById(arg0);
  }
  @PatchMapping(value = "/{id}")
  public Student updateEntity(Identifier arg0, @RequestBody() Student arg1) {
    return super.updateEntity(arg0, arg1);
  }
  @DeleteMapping(value = "/{id}")
  public void deleteEntityById(Identifier arg0) {
    super.deleteEntityById(arg0);
  }
  @Autowired()
  public StudentController(WebSocketHandler arg0) {
    super(arg0);
  }
}
```

Listing 1.4: The controller of the descendant `ModuleEnrolment` of `ModuleStudent`

```
@RestController()
@RequestMapping(value = "/students/{id}/enrolments")
public class StudentEnrolmentController extends DefaultNestedRestfulController<Student,
     Enrolment> {
  @GetMapping()
  public Page getInnerListByOuterId(Identifier arg0, PagingModel arg1) {
    return super.getInnerListByOuterId(arg0, arg1);
  }
  @PostMapping()
  public Enrolment createInner(Identifier arg0, @RequestBody() Enrolment arg1)
       {
    return super.createInner(arg0, arg1);
  }
  @Autowired()
  public StudentEnrolmentController(WebSocketHandler arg0) {
    super(arg0);
  }
}
```

Listing 1.5: The default service of `ModuleStudent`

```
@Service(value = "StudentService")
public class StudentService extends SimpleDomServiceAdapter<Student> {
  public Page getEntityListByPage(PagingModel arg0) {
    return super.getEntityListByPage(arg0);
  }
  public Student createEntity(Student arg0) {
    return super.createEntity(arg0);
  }
  public Student getEntityById(Identifier arg0) {
    return super.getEntityById(arg0);
  }
  public Student updateEntity(Identifier arg0, Student arg1) {
    return super.updateEntity(arg0, arg1);
  }
  public void deleteEntityById(Identifier arg0) {
    super.deleteEntityById(arg0);
  }
  public Collection getAllEntities() {
    return super.getAllEntities();
  }
  public void setOnCascadeUpdate(BiConsumer arg0) {
    super.setOnCascadeUpdate(arg0);
  }
  @Autowired()
  public StudentService(SoftwareImpl arg0) {
    super(arg0);
    this.setType(Student.class);
  }
}
```

□

5 Tool Support and Evaluation

5.1 Tool Support

We implemented our method as a module in our previously-developed software framework, named JDA [10]. Our implementation is available on GitHub[8]. The implementation uses the following third-party libraries and software:

– Java Parser [21] (version 3.8.2): used by both generator functions to parse and manipulate the Java source code.
– Frontend - React (version 17.0.2): used by `FEGen` to generate the `FESoftware` written in React library. This library is executed as part of the Node.js platform [1].
– In-memory Java compiler [14] (version 1.3): used by `BEGen` to compile Java code. The compiled code is executable by the framework.
– Reflections [18] (version 0.9.12): used by `BEGen` to collect the compiled Java code for execution.
– Backend - Spring Boot [17] (version 2.4.2): used by `BEGen` to generate the `BESoftware` written as a Spring Boot application.
– Database PostgresQL [2] (version 12.8): used by `BESoftware` to manage data created by in the software.

5.2 Performance Analysis

We analyse the time complexities of two generator functions: `FEGen` and `BEGen`. From Algs. 1 and 2, let us denote by $M = |\texttt{mccs}(s)|$ the number of modules of a software.

Proof. (*Function* `FEGen` *has quadratic time complexity.*) From Alg. 1, let us denote by $F = |c.\texttt{fields}|$ the number of domain fields of a domain class c.

The dominant part of Alg. 1 is a **foreach** loop which has M number of iterations. Each iteration makes three function calls. It is quite clear that function `genViewIndex` only operates on two fixed input and thus has a constant time. Among the other two functions: `genViewForm` and `genViewBrowse`, we would pick either one to analyse because there is not much difference between them in terms of execution time. Both functions use F domain fields to construct the view. A difference is that `genViewBrowse` additionally contructs a list of domain objects to present on generated table view. With the use of pagination, however, the this extra step is always performed in constant time.

Let us, thus, analyse function `genViewForm`, which is shown at the bottom of Alg. 1. The dominant part of this function is a **foreach** loop that has F number of iterations. Each iteration involves making two function calls, both of which operate on fixed input and thus are executed in constant times.

In brief, function `FEGen` has the worst-case time complexity of $M \times F$. It is a quadratic time, which is fast. □

[8] https://github.com/jdomainapp/jda-mosar

Proof. (*Function* `BEGen` *has quadratic time complexity.*) From Alg. 2, let us denote by $D = |T.\texttt{nodes} \setminus \{T.\texttt{root}\}|$ the number of non-root nodes of the containment tree T.

The dominant part of Alg. 2 is a **foreach** loop which has M number of iterations. Each iteration makes several function calls and executes a nested **foreach** loop. It is clear from the explanation about the algorithm that the function calls execute in constant times and, thus, the nested **foreach** loop is the dominant part of each iteration. This loop iterates over all the non-root nodes of T and so it has D number of iterations. Each iteration invokes the function `genDescendantControllerType`, which is also performed in constant time. Therefore, function `BEGen` has the worst-case time complexity of $M \times D$. It is also a quadratic time and fast. □

5.3 Discussion

Our evaluation is not without limitations. First, we focused mainly on the generator functions and not on the actual execution of the software. The execution time is also one indicator of the software quality. We are in progress of incorporating the software execution and deployment phases into our framework, which would allow us to easily execute the software after it has been generated. We will investigate this further in future work. Second, we were not able to compare our RFS generator functions' performance with those of other similar frameworks (e.g. Apache-IsIs [3], OpenXava [7] and jHipster [11]). There are two main reasons for this. First, there have been no performance data reported for any of these frameworks available in the public domain. Second, it is difficult to isolate the generator functions of these frameworks for analysis because they are executed as part of the software generation workflows.

6 Related Work

Our work in this paper is positioned at the intersections among three areas: SPD, DDD and RFS development.

SPD. Generative RFS development is a specialisation of the traditional SPD [4] for two components: frontend and backend software generation. Our work is unique in the use of a core domain model and a module-based software configuration to automatically generate both the frontend and backend. We support fullstack configuration whereby the frontend and backend are expressed in different language platforms. The generated RFS conforms to the MOSA^R architecture which we also defined in the paper.

DDD. Our work in this paper contributes to a current research theme [10,9] in enhancing the DDD method [5,22,12] with module-based software generation capabilities, especially using annotation-based DSLs [15]. Our contribution extends our most recent work [9] with an extended software architecture for RFS (MOSA^R) and a software generation method for this type of software. Comparing to recent DDD frameworks [3,7,11], our method is unique in using an incremental, module-based generation method. In this, software modules are first generated from the domain model and then combined to generate the software.

RFS development. Our method is similar to three DDD software frameworks ApacheIsIs [3], OpenXava [7] and jHipster [11] in the support for RFS software generation. Although these frameworks provide more RFS functionality than currently provided in our method, they do not follow an incremental module-based software generation approach as in ours. Consequently, developers of these frameworks cannot configure the module and software structure and behaviour before generation. In our method, developers can configure a module using MCCL and the software using SCCL. Further, this can be performed for both the frontend and backend software subsystems of RFS.

Apart from this main difference, ApacheIsIs and OpenXava do not separate the frontend and backend software as in our method. jHipster is more similar to our method in this regard. This framework takes as input a domain model, expressed in a DSL named JDL, and automatically generates and executes an RFS software. The target language platforms for both frontend and backend are configurable. However, the module's view generated by jHipster does not support nested views as in our method. This feature is necessary in some cases to show the domain objects of the associated domain classes in the domain model. It is supported in our method through the module containment tree. The JDL language only supports domain class specification. Similar to DCSL, jHipster's JDL also supports the domain field and associative field concepts. However, it is unclear whether its domain modelling DSL supports all the essential features as shown with our DCSL language. Further, JDL is an external DSL, which needs to be parsed separately from the target language platform. In contrast, DCSL is an internal DSL and does not require a separate parser.

7 Conclusion

In this paper, we proposed a solution to address the problem of generative, module-based RESTful full stack software construction. We adopted our incremental, module-based DDD approach and MOSA architecture to define an architecture named MOSA^R for RFS. We extended our work on generative software development in MOSA to perform RFS generation in MOSA^R. The enhanced generator includes two components: one for the frontend software and the other for the backend one. Both software subsystems can be configured and executed by different programming platforms. We defined an RFS software configuration to address the core requirements of RFS generation in MOSA^R. We presented two algorithms for the frontend and backend sofware generation functions. The frontend function, in particular, generates the frontend modules following the popular single-page component design. We evaluated both algorithms to show that they are scalable of generating large software.

Our method is novel in its incremental module-based DDD software generation approach, which generates the software module configurations first from the domain model and then uses them to generate configurations for different software variants. The developers and domain experts can be involved in each step to customise the generated configurations so that they match the domain requirements.

We contend that our contributions in this paper constitute a novel contribution to both the DDD method and the RESTful full stack software development community. In future work, we plan to enhance our implementation to support other frontend and backend technologies. We would also like to extend our design further to tackle other full stack challenges [20] posed by modern Internet technologies, including micro-services architecture and cloud and serverless computings.

Acknowledgement

The authors would like to thank Binh Hai Do for his physical design and implementation of the first prototype of the RFS generation functions using React.js and Spring Boot. These were carried out in Spring 2021 as part of his graduation thesis project at the Faculty of IT, Hanoi University.

References

1. Node.js (2021), https://nodejs.org
2. PostgreSQL (2021), https://www.postgresql.org/
3. Apache-S.F: Apache Isis (2021), http://isis.apache.org/
4. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013)
5. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional (2004)
6. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. ACM Trans. Internet Technol. **2**(2), 115–150 (May 2002)
7. Javier Paniza: OpenXava (2021), http://openxava.org/
8. Le, D.M., Dang, D.H., Nguyen, V.H.: On Domain Driven Design Using Annotation-Based Domain Specific Language. Computer Languages, Systems & Structures **54**, 199–235 (2018)
9. Le, D.M., Dang, D.H., Nguyen, V.H.: Generative Software Module Development for Domain-Driven Design with Annotation-Based Domain Specific Language. Information and Software Technology **120**, 106–239 (Apr 2020)
10. Le, D.M., Dang, D.H., Vu, H.T.: jDomainApp: A Module-Based Domain-Driven Software Framework. In: Proc. 10th Int. Symp. on Information and Communication Technology (SOICT). pp. 399–406. ACM (2019)
11. Matt Raible: JHipster - Full Stack Platform for the Modern Developer! (2021), https://www.jhipster.tech/
12. Millett, S., Tune, N.: Patterns, Principles, and Practices of Domain-Driven Design. John Wiley & Sons (Apr 2015)
13. Nair, V.: Practical Domain-Driven Design in Enterprise Java: Using Jakarta EE, Eclipse MicroProfile, Spring Boot, and the Axon Framework. Apress, 1st edn. (Sep 2019)

14. Nguyen, T.: InMemoryJavaCompiler (Oct 2017), https://github.com/trung/InMemoryJavaCompiler, original-date: 2015-03-05T14:02:26Z
15. Nosál', M., Sulír, M., Juhár, J.: Language Composition Using Source Code Annotations. Computer Science and Information Systems **13**(3), 707–729 (2016)
16. OMG: Unified Modeling Language version 2.5 (2015), http://www.omg.org/spec/UML/2.5/
17. Pivotal: Spring Boot (Oct 2017), https://spring.io/projects/spring-boot
18. Ronmamo: Reflections (2020), http://github.com/ronmamo/reflections
19. Scott, E.: SPA Design and Architecture: Understanding Single Page Web Applications. Manning, Shelter Island, NY, 1st edn. (Nov 2015)
20. Taivalsaari, A., Mikkonen, T., Pautasso, C., Systä, K.: Full Stack Is Not What It Used to Be. In: Brambilla, M., Chbeir, R., Frasincar, F., Manolescu, I. (eds.) Proc. Web Engineering. pp. 363–371. LNCS, Springer Int. Publishing, Cham (2021)
21. Tomassetti, F., Bruggen, D.v., Smith, N.: JavaParser (2021), http://javaparser.org/
22. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley Professional, Upper Saddle River, NJ, 1st edn. (Feb 2013)
23. Zimarev, A.: Hands-On Domain-Driven Design with .NET Core: Tackling complexity in the heart of software by putting DDD principles into practice. Packt Publishing (Apr 2019)