Bridging Programming Productivity, Expressiveness, and Applicability: a Domain Engineering Approach

Oded Kramer and Arnon Sturm
Department of Information Systems Engineering, Ben-Gurion University of the Negev
Beer-Sheva, Israel
odedkr@bgu.ac.il, sturm@bgu.ac.il

Abstract. Productivity is the ability to create a quality software product in a limited period with limited resources. The software engineering community advocates that the future of productivity lies in the field of domain engineering. However, existing domain engineering approaches suffer from the tension between productivity and applicability. In this paper we propose an approach that reduces this tension by adopting a domain engineering method called Application-based DOmain Modeling (ADOM) as an infrastructure for a new programming approach. The adopted ADOM is applied on Java as its underlying language. This approach will offer guidance and validation for application developers as mechanisms for improving their productivity. This is done by keeping the regular Java development environment and thus maintaining the developer's expressiveness and not compromising the overall applicability of the approach.

Keywords: Domain engineering, software productivity,

1 Introduction

Today's software development is a complex process involving a set of activities that require orchestration. One of the most resource consuming activities is programming. In order to better utilize the programming activity we should seek for ways to increase its productivity. Productivity according to [13] is "the ability to create a quality software product within a limited period with limited resources". The productivity of a programmer is affected by many factors. Jones [8] presented several of these: the design for reusability, experience, bugs or errors, management, creeping requirements, code structure and complexity, application size, supportive tools, and programming languages.

Many efforts have been made in order to increase the programmers' productivity from the technical point of view. These efforts are focused on providing techniques for increasing the code reusability, thus saving programming time. These techniques include generic programming which enables reuse by parameterizations, design patterns which provide solutions for specific situations, meta programming which enables programming at various levels of abstraction, as well as utilizing reflection

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.): CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 47-60, 2010.

2.1. DSLs

Domain Specific Languages (DSLs) are computer languages that are tailored to specific domains [11, 14]. This reduction to a specific domain allows the elevation of the language abstraction level. A higher abstraction level is a sought out goal in the fields of DSLs [6, 9, 11, 14]. It leads to many benefits such as: increased productivity, improved quality, better maintainability, and reuse of experts' knowledge. DSLs are divided into two distinct types: external and internal DSLs.

2.1.1. External DSL

The basic premise of external DSLs is that the underlying principles of a higher abstraction level and tailoring to specific domain necessitate the development of the DSL from scratch. Typically, there would be a domain expert whose expertise is on the semantics of the domain and an expert programmer whose expertise is on developing complicated and sophisticated software[1] working on this process [11]. The design process includes defining domain concepts and their relationships, semantics, notations, and constraints. The implementation process includes building a code generator, an optional domain specific framework, and the DSL's integrated development environment (IDE) which includes the DSL's supporting tools.

The main two advantages of external DSLs are the improved productivity; reports have shown of increase in productivity of 300%-1000% [11] and enhanced application quality; due to a preliminary check of the model's consistency according to domain rules. This means that many of the programmers' mistakes can be detected and thus can be avoided at this early stage of development. The developers specify the solution on a higher level, which is then transformed automatically to another form of code. This means that they can avoid dealing with important but complicated issues such as design principles and architecture, as these are handled by the code generator.

Yet, external DSLs suffer from various limitations. As mentioned, the design and implementation of external DSLs is by no means simple, it is complicated and time consuming. Even if the work is done by experts (both domain and programming), and some supporting tools are available it might not be enough to ensure a successful working DSL. According to [6] most DSLs are usually abandoned in the development process and the work is done eventually in regular general purpose languages. Additionally, to justify economically the investment of the DSL development process a quota of applications has to be exceeded. While this is true for all domain engineering techniques it is as harsh as the amount of emphasis that is put on the domain engineering process [6, 9]. Moreover, introducing the notion of DSL based development into an organization requires a significant change in the organization's development paradigm. This change requires both new tools and new processes. While some managers will be able to see the long terms advantages of DSLs, other might be reluctant to introduce radical, expensive and time consuming changes to

[1] obviously, they could be the same person, however both kinds of expertise are required

programmers can use the API in any desirable way. Thus, in that sense internal DSLs are less productive than external DSLs.

2.2. Feature-oriented approaches

Feature oriented approaches rely on features which are system properties that are relevant to the stakeholders and are used to capture commonalities or discriminate among systems in a product family [3]. The various approaches consist of a feature model that contains all features covered by the product family along with their dependencies and their variability [15]. Each application will be comprised by a unique subset of the features presented in the feature model. Typically, the feature model will be expressed using the tree diagram that was firstly introduced by the Feature-Oriented Domain Analysis (FODA) method [10].

The different feature oriented approaches focus on different levels of abstractions and on different stages of the development cycle. For example: FODA focuses on the domain analysis phase, Hyper/UML [15] and the work presented in [5] focus on feature oriented design by mapping features to other models (e.g., UML models). Feature Oriented Programming (FOP) [1] and HyperJ [18] focus on mapping features to code increments.

Many feature-oriented approaches suffer from the tension that was presented in the previous section. For example, feature modeling can help facilitate DSL design and DSLs may be used to specify the family members [4]. In that case, the applicability of the feature-oriented approach is problematic, similarly to that of the external DSLs. Furthermore, some approaches limit the expressiveness of application developers to the extent of only selecting appropriate features that are mapped automatically to code pieces (e.g., FOP and HyperJ [15, 18]). These, as in external DSLs, also may suffer from extensive domain engineering efforts, radical changes to the programming paradigm and narrow domains which will presumably lead to poor applicability.

To overcome the aforementioned limitations with respect to the tension between productivity and applicability, we utilize a domain engineering approach called Application-based Domain Modeling (ADOM).

3 The ADOM Approach

The Application-based Domain Modeling (ADOM) is rooted in the domain engineering discipline [16, 17], which is concerned with building reusable assets on the one hand, and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and validation of the enterprise-specific models against the relevant reference models.

The architecture of ADOM is based on three layers: The ***language layer*** comprises metamodels and specifications of the used languages. The ***domain layer*** holds the building elements of the domain and the relations among them. It consists of

multiplicity indicator is used to constrain the domain's applications to have classes classified as someDomainClass at least A times and no more than B times. This type of constraints in ADOM is referred to as the multiplicity constraint. In the application layer the someClassApplication class is classified by the someDomainClass class.

```
// domain layer code
@multiplicity(min = A, max = B)
public class someDomainClass {
    …
}
// application layer code
@someDomainClass
public class someApplicationClass {
    …
}
```

Listing 1: The Java annotation classifications

4.1. Structural constraints

Using the multiplicity indicator one can express a great deal of the structural commonality and variability captured and identified in the domain. For example, small scale information systems based on three layered architecture may be considered as a domain.
Applications in that domain use a relational DBMS, the JDBC API to interface with it, and the Java Swing API for the presentation layer. Applications in that domain may include a conference management system, a university registration system, and a laboratory management system.

In Figure 1, the applications of a conference management system and a laboratory management system are depicted along with their corresponding domain[2]. In this case the domain layer consists of five different types of classes GUI, Controller, and DBmapper, which represent the three classic layers, and SingleStatedObject and, MultiStatedObject which represent domain elements that have a single state or more, respectively.

In Listing 2, it is shown that the applications are expected to have exactly one class classified as a controller. This is indicated by the multiplicity annotation assigned to the class declaration as noted above. Moreover, it is shown that this class must have exactly one field classified as db, which is of a type that is classified as a DBmapper. This is noted by the DBmapper type of db in the domain code. This is effectively the composition relationship between these two classes that is shown in Figure 1. If the matching application field will be of any other type it will be a

[2] Note that domain models in ADOM-Java are expressed in Java. In Figure 1 we use UML to visualize the structural outline that was extracted.

language constraint. Others can be used to express that methods can be of a combination of different access levels, final or non-final, and static or non-static. Actually ADOM-Java supports all the Cartesian products of the different members' modifiers.

```
//domain layer code
@Multiplicity(min = 1, max = 1)
public class Controller {

 @Multiplicity(min = 1, max = 1)
  DBmapper db;

@Multiplicity(min = 1)
 public SingleStatedObject addDomObect(String...
ObjectsData)

@Multiplicity (min = 1)
 public boolean addDomainAssociation ()

@Multiplicity(min = 1)
public boolean changeStatDomObj(MultiStatedObject mso) }
```

Listing 2: The controller class in the domain layer

```
//domain layer code
@typing ({" SingleStatedObject ", " MultiStatedObject "})
@Multiplicity(min = 1)
 public singleStatedObject addDomObect(String…
ObjectsData)
```

Listing 3: the addDomObject method from Listing 2 with the typing indicator

Listings 2 and 3 are neither a complete description of the entire domain model, as the other 4 classes from Figure 1 are missing, nor a complete description of the entire controller domain class. The full implementation of this class has more methods and goes into the methods declarations themselves.

The matching application code regarding its controller class from the labs management system is presented in Listing 4.

First of all, the AppController class is classified as the Controller class from the domain; this is noted by the Controller annotation assigned to the class declaration. Following there is a field declaration which is classified as the db field from Listing 2. If the AppMapper type will be classified as DBmapper (not shown here) the language constraint will be fulfilled. Following, there are two methods declarations classified as addDomObect. Their public, non-final, and non-static modifiers indicate an adherence to the language constraint in Listing 2. Their return types' classifications are not

introduced to behavioral aspects as well. This will be shown by yet another drill down, this time to the controller's changeStatDomObj method as appears in Listing 5.

```
//domain layer code
@Multiplicity(min = 1)
public boolean changeStatDomObj(
@Multiplicity(min = 1, max = 1) MultiStatedObject mso) {

   @Multiplicity(min = 1, max = 1)
   if (dso.changeState()) {
        @Multiplicity(min = 1, max = 1 )
       db.updateDomainObject(dso);
           @Multiplicity(min = 1, max = 1)
            return true;
        }
   @Multiplicity(min = 1, max = 1 )
   return false;
}
```

Listing 5: The controller's changeStatDomObj method

First of all, this Listing presents this method's signature as it appeared in Listing 2 with the addition of the multiplicity indicator to the received parameter. This method's responsibility is to receive a business logic object, to change its internal state, update the DB if the transition was successful, and finally to return a Boolean statement indicating whether the action was successful or not. For this reason, the matching application methods will have to receive a single parameter classified as MultiStatedObject.

This is noted by the type of the mso parameter and by its multiplicity. Therefore, this example illustrates that constraints over methods' parameters can be defined in the same manner as over classes' fields. Following, inside the body of the method, there are four execution statements, each with a multiplicity[3] indicator constraining the statement to appear once. This specification constrains any application method classified as changeStatDomObj to contain each of these four statements exactly in the order as they appeared in the domain and with the same scoping structure, with the exception of method calls. Each method call in the domain will be replaced in the application code by a call to a method that is classified as the called method in the domain. For example, dso.changeState() method call in Listing 5 is replaced by the p.accept call in Listing 6. This is correct only because p is of type Paper (as presented in Figure 1), which is classified as statedObject and accept() is a call to its method

[3] Notice that this use of Java annotation is not supported in standard Java and requires an extension called @Java [2].

demonstrated in Listing 3, rather that the type itself will be Connection. In fact, this is a specification of how to interface with JDBC; in this case to (re)use by composition of the Connection class

An application class that will be classified as DBmapper will violate the specification in Listing 7 if it doesn't have a single connection field of type Connection. ADOM-java offers an additional way to constraint APIs extension. This is referred to as the extension constraints. The usage of some APIs can be a quite a difficult task [18]. This has many reasons. For example, the volume of some of the APIs can be overwhelming (the Swing API has hundreds of classes). Moreover, inheriting from a framework necessitates an understanding of its inner structure. This can become quite difficult as the interdependencies of the classes force developers to learn all the classes at once rather than each class at a time. ADOM-Java realizes that for some domains only a small subset of the API will suffice. For example, of the entire Swing API only a dozen classes are used in the aforementioned applications. Here lies the motivation for the extension constraint (not shown here). It will be used to define if a framework class can be extended in the application and by which mechanism, where the possible mechanisms are: composition, inheritance, and none. For example, some Swing components can be found too complicated or unnecessary thus can be marked as not to be used for a given domain at all, others can be marked as not to be extended (i.e., used only by composition).

5 Summary

In this paper we presented the tension between productivity and applicability in common domain engineering approaches. We pointed that a key factor for reducing this tension is the expressiveness of the application developer. To address this tension, we utilize a domain engineering approach called ADOM based on the Java programming language for guiding the application developer by providing models that express the expected structure and behavior of the domain's applications. Moreover, ADOM-Java validates the developer's code according to these models. Thus, it enables error detections at an early stage of development. These factors, presumably will lead to increased productivity. Furthermore, ADOM-Java is embedded into a general purposed programming language (Java), thus it ensures that the expressiveness of the application developer will not be compromised and that the overall approach, as it does not necessitate radical expansive changes to the programming paradigm, will be applicable.

While ADOM-Java looks promising in bridging the gap between productivity, expressiveness and applicability. It is clear that additional examination is required. In the near future, we plan to conduct and experiment that aims at checking the applicability of ADOM-Java.