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.

mechanisms, and frameworks which provide partial design and implementations but are difficult to compose [3]. However, most of these efforts are related to general purpose reuse techniques, thus they do not exploit the commonalities among similar applications of a given domain.

Nowadays, the software engineering community advocates that the future of productivity lies in the field of domain engineering [3, 4, 11, 19]. According to Haarsu [7], domain engineering is a systematic process to provide common core architecture for similar applications. Its purpose is to provide reuse capabilities among these applications.

Indeed significant productivity achievements have already been reported [11, 12], but the quest for better software development solutions is far from over. We claim that one of the reasons for this is the inherent tension between productivity and applicability that current domain engineering approaches suffer from. Solutions that offer potentially promising productivity results tend to be expensive and require radical changes to the accustomed programming paradigms, thus their applicability is low.

A key factor that can aid in resolving this tension is expressiveness, which is the ability of developers to express desired semantics. Expressiveness is highly correlated with applicability. Solutions that reduce significantly the developer's expressiveness often require new development tools and processes. These tend to be expensive and require a learning curve that might seem to managers as risks that should be avoided. We assert that desired solutions should strive to keep the level of expressiveness as in general-purpose languages.

In this paper we propose an approach that aims at partially resolving the above mentioned tension by adopting a domain engineering method called Application-based DOmain Modeling [16, 17] (ADOM) as an infrastructure for a new programming approach. This approach offers guidance and validation for application developers as mechanisms for improving their productivity. The novelty of the proposed approach lies in using a standard programming language (including it supporting tools), thus maintaining the developer's expressiveness and increasing the applicability of the approach.

The structure of the rest of the paper is as follows. Section 2 discusses related work concerning DSLs and feature-oriented programming approaches, delving into the tension between productivity and applicability. Section 3 briefly introduces ADOM - the underlining framework of the proposed approach, which is presented in details in Section 4. Finally, Section 5 concludes and refers to future research directions.

2 Related Work

As domain engineering provides the platform for increasing productivity, in this section we analyze domain specific languages and feature-oriented approaches in view of the above mentioned tension.

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

their natural development process. All of these indicate that the applicability of external DSLs is problematic.

Another limitation of external DSLs is the limited expressiveness of the application developer. Usually, this is considered to be an advantage – limiting the application developer's expressiveness means guiding him and controlling the quality of his work, and by that increasing his productivity and the overall quality of the end products. However, we consider this to be a disadvantage since the application developer has many constraints. The restrictions imposed on application developers are achieved by designating the domain to include a set of pre formulated commonality and variability. In case the application developers wish to express a newly encountered feature, they have to inform the DSL developers to update the DSL and wait for the change to be done. This process is time consuming and more importantly will make the procedure of incorporating new variants into the domain difficult, ultimately leading to narrow domains. Furthermore, this limitation is directly linked to the necessity to incorporate new tools and processes which lead to the problematic applicability of the DSL.

2.1.2. Internal DSLs

Internal DSLs drew their inspiration from the recognized drawbacks of external DSLs. Their basic premise is that DSLs should not be developed from scratch; rather they should be embedded on existing proven general purposed programming languages (GPPLs). In this sense internal DSLs are no different than regular domain specific application programming interfaces. However, they are different in the sense that the APIs are designed to have a language like flow to them. This is achieved by advanced coding techniques such as method chaining, expression builders, interface chaining, generics, etc. When these techniques are used correctly some domain semantics could be validated in compile time.

The main advantage of internal DSLs is that they do not suffer from the above mentioned drawbacks of external DSLs. This is caused by three main reasons: (1) The development of internal DSLs is much easier with respect to external DSLs, mainly because the GPPL facilities already exist; (2) Internal DSLs do not necessitate a radical change in the organization's natural development paradigm as they permit using the same set of tools (such as a programming languages, IDEs, and compilers); and (3) Internal DSLs do not limit application developers' expressiveness as they are allowed to use the GPPL regularly. These reasons indicate that internal DSLs are more applicable than external DSLs.

However, internal DSLs introduce the following limitations: (1) Current reports [6, 9] of internal DSLs focused on code readability and maintainability. Although this should have positive effects over productivity it is hard to see how sophisticated APIs raise the level of abstraction similarly to external DSLs; (2) External DSLs achieved improved code quality through pre code generating validation algorithms and higher abstraction levels. Although internal DSLs can exploit coding techniques to assure some domain semantics, they cannot implement validation algorithms that examine the specified code according to domain constraints. Ultimately, application

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

specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The structure and the behavior of the domain layer are modeled using the language that was defined in the language layer. The ***application layer*** consists of domain-specific applications, including their structure and behavior. The application layer is specified using the knowledge and constraints presented in the domain layer and the constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain should be applied in any application of that domain. In order to achieve this goal, any element in the application is classified according to the elements declared in the domain.

For describing variability and commonality, ADOM uses a multiplicity indicator that can be associated to all elements, including classes, attributes, methods, and more. The multiplicity indicators in the domain aim to represent how many times an element of this type may appear in an application. This indicator has two associated tagged values - min and max - which define the lowest and the upper most multiplicity boundaries.

The relations between a generic (domain) element and its specific (application) counterparts are maintained by a classification mechanism: each one of the elements that appear in the domain can serve as a classifier of an application element of the same type (e.g., a class that appears in a domain may serve as a classifier of classes in an application). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain. Some optional generic elements may be omitted and not be included in the application, while some new specific elements may be inserted in the specific application; these are termed application-specific elements and are not classified in the application.

ADOM also provides validation mechanism that prevents application developers from violating domain constraints while (re)using the domain artifacts in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application in order to fulfill particular application requirements.

While ADOM is general and language-independent, a specific language needs to be selected as a basis for a workable dialect of ADOM. In order to apply ADOM, the only requirement from the associated language is to have a classification mechanism that enables categorization of elements.

4 The ADOM-Java Dialect

Since we refer to programming, in this paper we select Java as the language used in conjunction with ADOM. We will refer to that ADOM dialect as ADOM-Java. In this case, the required classification mechanism will be fulfilled by Java's annotation construct due to its meta data qualities. Listing 1 demonstrates the usage of the Java annotation in both the domain and application layers. In the domain layer the

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.

violation of the code presented above. This type of constraints goes one step further than the multiplicity constraint as it deals with the syntactic structure of the application, for this reason it is referred to as the language constraint.

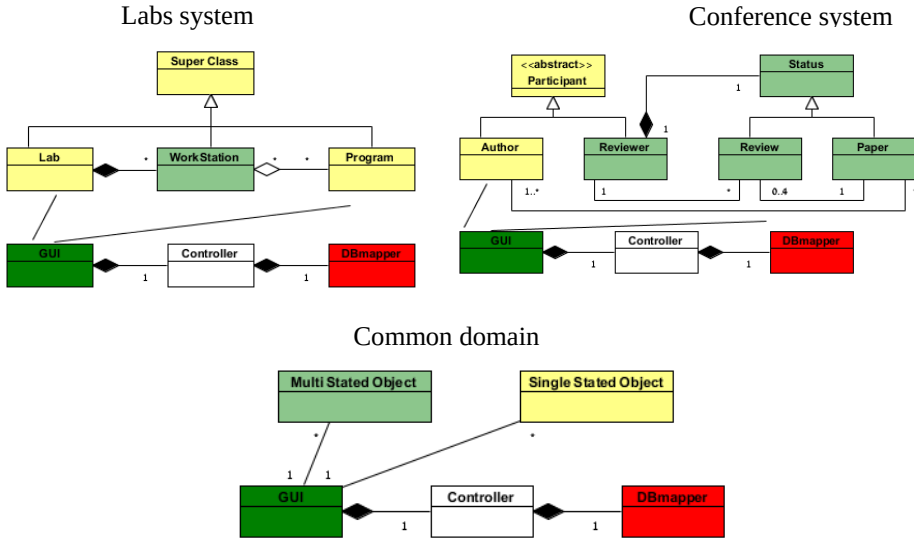

Figure 1: example of two applications and their common domain

The rest of Listing 2 expresses three methods, each of which are expected to appear at least once in the controller class. All of these methods should be pubic, the last two should return the boolean primitive type, and the first one should return a type classified as SingleStatedObject. These constraints are indicated by the methods' modifiers. Any other setting will be a violation of the language constraint. These method attributes are stated explicitly, however, those that are stated implicitly will be constrained by the language constraint as well. Effectively, all of the application methods classified as one of this three will have to be non-static and non-final, moreover the field that will be classified as db will have to be private-package.

ADOM-java enables to use other indicators to raise the flexibility of the language constraint. For example, the addDomObject method which is responsible of adding new objects to the system and returns the newly added object should be able to return both the SingleStatedObject and the MultiStatedObject types. This requires a correction to the code represented in Listing 2 as shown in Listing 3.

The typing indicator expresses this. It should be noted that the indicator overrides the return type. This was used because there are no multiple return types in Java. It is important to notice that this is not to say that the respected application method will return both types, indeed it will return a single type, as accustomed in Java. However, this type will be either classified as SingleStatedObject or as MultiStatedObject. This is just one example of additional indicators that raise the flexibility level of the

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

shown here as well, however in the full implementations they are of type SingleStatedObject and MultiStatedObject and therefore correct. These methods are responsible for adding two (Lab and Program) of the three classes that were presented in Figure 1 to the labs management system. The number of these methods indicates an adherence to the multiplicity constraint in Listing 2. Following, there is a method classified as changeStatDomObj which adheres to both the multiplicity and the language constraints on Listing 2. Finally, there is the reportMalfuctionWS method, which is responsible of changing the state of workstation from functional to malfunction, removing the workstation from its lab and updating the DB if the change took place, and which has no classification. This method does not match any of the domain's method types; therefore it can be of any linguistic structure and multiplicity, and is considered as an application specific extension.

```
//application layer code
@Controller
Public class AppController {

   @db
   AppMapper appDB;

   @addDomObject
   public Lab addLab()
   @addDomObject
   public Program addProgram()

   @changeStatDomObj
   public boolean fixWS(WorkStation ws)

   public boolean reportMalfucntionWS(WorkStation ws)
```

Listing 4: The matching application code with respect to Listings 2 and 3

The same goes for all other Java constructs: classes, fields, etc. Obviously, the code presented in Listing 4 is not a complete description of the application's controller implementation. Some method declarations were omitted and the bodies of the methods were not presented due to space limitations. These will be presented and elaborated in the next section

4.2. Behavioral constraints

In the previous section we presented how the multiplicity and language constraints are used to impose structural knowledge over the applications. Language constraints by nature cannot be extended to behavioral knowledge as they refer to syntactic structure of the Java constructs. However, the notion of multiplicity constraints can be

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].

that is classified as changeState(). Thus, the code in Listing 6 adheres to behavioral constraints specified on Listing 5.

```
//application layer code
@changeStatDomObj
 public boolean acceptPaper(Paper p)  {
        if (p.accept()) {
            db.updatePaperStatus(p);
            return true;
        }
        return false;
 }
```

Listing 6: an application method that adheres to the method presented in Listing 6

The example in Listing 6 presents an application method that did not introduce application specific statements. These statements could have been introduced anywhere in method body as long as the constraint mentioned above would not have been violated.

4.3. Extension constraints

Up until now we presented two different kinds of types: primitive types that are part of the language, and domain classified types, which means types that are classified as one of the classes from the domain layer (Figure 1 presents these). However, there is a third kind, types that belong to horizontal domains which are parts of software systems that can be classified according to their functionality [4]. Examples of these types are those from the Swing, JDBC, and collection APIs. Listing 7 presents the DBmapper class (figure 1) which uses this kind of types.

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

    @Multiplicity(min = 1, max = 1)
    Connection connection;
}
```

Listing 7: Horizontal domain types

This Listing specifies that a matching application class will have a single field named connection of type Connection (a type from the JDBC API). This is not to say that the Application's respected field will be of a type classified as connection, as was

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.

6 References

1. Batory, D., Sarvela, J.N., and Rauschmayer, A. Scaling step-wise refinement. Proceedings of the 25th International Conference on Software Engineering, 187–197, 2003.
2. Cazzola, W. @Java: A Java Annotation extension, http://homes.dico.unimi.it/~cazzola/atjava.html, 2010.
3. Czarnecki, K. and Eisenecker, U. W. *Generative Programming - Methods, Tools, and Applications*, Addison-Wesley, 2000.
4. Czarnecki, K. Overview of Generative Software Development. Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, September, 15–17, 2004.
5. Czarnecki, K. Mapping features to models: A template approach based on superimposed variants, in GPCE'05, Lecture Notes in Computer Science 3676, 422-437, 2005.
6. Freeman, S. and Pryce, N. Evolving an embedded domain-specific language in Java. In Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, 855-865, 2006.
7. Harsu, M. A survey on domain engineering, Report 31, Institute of Software Systems, Tampere University of Technology, 2002.
8. Jones, C. *Estimating Software Costs*, McGraw-Hill, 2007.
9. Kabanov, J. and Raudjärv, R. Embedded typesafe domain specific languages for Java. Proceedings of the 6th international Symposium on Principles and Practice of Programming in Java, 189-197, 2008.
10. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., 1990.
11. Kelly, S. and Tolvanen, J-P. Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008.
12. Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. A software engineering experiment in software component generation. Proceedings of the 18th international Conference on Software Engineering, 542-552, 1996.
13. Lowell, A. J. *Programmer Productivity: Myths, Methods, and Murphology. A Guide for Managers, Analysts, and Programmers.* John Wiley and Sons, 1983.
14. Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. ACM Comput. Survev. 37 (4), 316-344, 2005.
15. Philippow, I., Riebisch, M., and Boellert, K. The hyper/UML approach for feature based software design. The 4th AOSD Modeling With UML Workshop, 2003.
16. Reinhartz-Berger, I. and Sturm, A. Enhancing UML Models: A Domain Analysis Approach, Journal on Database Management, 19 (1), special issue on UML Topics, 74-94, 2007.
17. Reinhartz-Berger, I., Sturm, A. Utilizing Domain Models for Application Design and Validation. Information and Software Technology, 51(8), pp. 1275-1289, 2009.
18. Tarr, P. and Ossher, H.: Hyper/J User and Installation Manual. In: Multi-Dimensional separation of Concerns: Software Engineering using Hyperspaces, 2001.
19. Weiss, D. M. and Tau, C., and Lai, R. *Software Product Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, 1999.