

Utilizing Application Frameworks: A Domain Engineering Approach

Arnon Sturm and Oded Kramer

Abstract Application frameworks aim to provide coherent code to be used and reused. The primary benefits of application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers. Yet, as these frameworks become more extensive and complex, their usage becomes a burden and requires further effort. In this chapter we adopt the Application-based DDomain Modeling (ADOM), a domain engineering approach offering guidance and validation for developers when using existing knowledge, as in the case of application frameworks. The approach is adopted in the context of a programming language and demonstrated with the use of Java and is thus denoted as ADOM-JAVA. The approach preserves the regular development environment and requires minimal adaptation for using the proposed approach. We also demonstrate the use of ADOM-JAVA as a vehicle for defining and using domain-specific languages. Finally, we evaluate the use of ADOM when applied to a Java-based development. Following the guidance and validation capabilities provided by the proposed approach, the experiment shows that the productivity of the developers in terms of time and quality is expected to increase.

Keywords Application framework • Domain engineering • Reuse • Software composition

A. Sturm (✉) • O. Kramer

Department of Information Systems Engineering, Ben-Gurion University of the Negev,
Beer-Sheva, Israel

e-mail: sturm@bgu.ac.il; odedkr@bgu.ac.il

1 Introduction

Software development is a process that may involve the reuse of readymade and tested artifacts, such as components, executables, libraries, and frameworks. The formalization of these approaches has pervaded various application and research areas, such as software product line engineering (SPLE) [5, 24]. In SPLE there are two main phases: (1) the domain engineering phase in which the knowledge encapsulated within a domain is specified with the aim of being reused and (2) the application engineering phase in which the domain knowledge is reused and adapted as required by specific applications. To specify the reusability of the various artifacts, the software engineering community devised techniques that include generic implementation techniques for increasing software artifacts (such as models, components, and code) reuse. These techniques include generic programming that enables reuse by parameterizations, design patterns that provide solutions for specific situations, meta programming that enables programming at various levels of abstraction, as well as utilizing reflection mechanisms, and frameworks [6]. Although the design for reuse is a key issue, one should also provide mechanisms to better facilitate that reuse in an effective and productive manner. Indeed, existing tools and approaches support reuse specification at the design level; however, these mainly focus on instantiation and configuration. Moreover, when referring to programming and code, it seems that the guidance provided by the various techniques is limited and mainly provides a means for specifying reuse without proper guidance on how it should actually be performed.

In this chapter we propose an approach that aims at guiding the reuse of software frameworks (i.e., code) by adopting a domain engineering method called Application-based DOMain Modeling (ADOM) [25, 26] as an infrastructure for a new programming approach. In general, ADOM supports the reuse of many software artifacts. Nevertheless, in this chapter we apply it to a specific type of software artifact—code. We term the new approach ADOM-JAVA as we apply ADOM in the context of the Java programming language. This approach offers guidance and validation for application developers that better facilitates the domain knowledge and code reusability. The uniqueness of the proposed approach lies in the utilization of a standard programming language (including its supporting tools), and thus keeps developers within their standard development environment and the explicit reuse guidance provided within the domain knowledge.

The structure of the rest of the chapter is as follows. First, we discuss related work concerning framework usage and the reuse types. The following section presents the java agent development (JADE) framework, which is widely used for developing multi-agent systems (MAS), as a case study and a demonstrator for the problems and solutions of utilizing application frameworks. Next, we introduce ADOM—the underlining framework of the proposed approach, followed by a description of the actual application of the ADOM approach in the context of JADE and Java as the used language. To further demonstrate the use of the proposed approach, we discuss its utilization in the context of domain-specific languages. Having set the

details of applying ADOM-JAVA, we report on an initial evaluation we performed. Finally, we conclude and discuss future research directions.

2 Related Work

2.1 Software Frameworks

Software frameworks are “semi complete applications that can be specialized to produce custom applications” [9]. These frameworks are widely used in software development in general [22] and in domain-specific areas, in particular [10]. Fayad and Schmidt classified the various frameworks by “the techniques used to extend them, which range along a continuum from whitebox frameworks to blackbox frameworks” [8]. Whitebox frameworks support the reuse of their functionality by inheriting base classes and overriding predefined methods and require the developers to be familiar with their internal structure. On the other hand, blackbox frameworks support the reuse of their functionality by defining components, which meet the interface requirements and integrating these components into the specific framework and are mostly used by using object composition. Note that the whitebox frameworks are more commonly used, as developing blackbox frameworks requires much effort. Fayad and Schmidt further stress the importance of frameworks as a means for code reuse. Following their analysis, it seems that frameworks capture most of the principles provided by other approaches such as patterns, class libraries, and components. However, when constructing frameworks one should address a number of challenges, namely, *development effort*, *learning curve*, *integratability*, *maintainability*, *validation and defect removal*, *efficiency*, and *lack of standards* [8].

As determined by [22], although it is widely agreed that framework-based development improves productivity (in terms of development time and code quality), many frameworks still suffer from limited or wrong usage. This indicates that there is a need for further improvements in this kind of development. Polancic et al. [22, 23] examined the causes for this situation. They found out that the acceptance of frameworks is mainly dependent on two factors: continuous framework usage intention and the perceived usefulness of the framework. Also, their results indicate the understandability, which is “the capability of a software product to enable the user to understand whether the software is suitable and how it can be used for particular tasks and conditions of use” [14] is a major factor in using frameworks. This finding is in line with the work of Ali et al. [1], which found that applications developed by novice software developers based on provided frameworks resulted in poor software quality. This was somewhat surprising as all developers (students) seem to have abandoned the theory they were taught regarding design principles.

Summarizing the notion of framework usage, improvements are required in providing further guidance for reusing whitebox frameworks. Next, we elaborate on reuse mechanisms that can be used for guiding the aforementioned reusability.

2.2 Reuse Mechanisms

The notion of reuse has evolved over many years. Becker et al. [2] classify the reuse area into various approaches: Patterns, which define (general) templates to solve commonly occurring problems; components, which can be used and composed as is; modules, the abstract objects, which have to be instantiated to be of concrete use; and reference models (RM), which comprise information that suits various situations. In the context of this chapter, we refer to frameworks as reference models, in the sense that they fit many situations (applications). Below we describe various reuse mechanisms based on the studies of Becker et al. [2], vom Brocke [3] and Jacobson et al. [15] (among others) and present these in the context of reusing elements from the frameworks for developing applications. To demonstrate these reuse mechanisms we use a control system framework consisting of abstract concepts such as sensor, controlled element, controlled value, etc.

- *Analogy Construction*: An analogy means the transfer of information or implementation from the framework to the application, enabling the required adaptation. For example, in the case of the control system framework one can make the analogy from a sensor within the framework, to a thermometer within a climate control system or to a smoke detector which is a part of an alarm system.
- *Aggregation*: Aggregation means that one can assemble parts from the framework to construct a specific application. In the case of the control system framework, the climate control system or the alarm control system can be assembled by adopting only the relevant framework parts, such as sensors, and controlled values, neglecting other parts of the framework.
- *Configuration*: Configuration means the modification of certain framework elements following predefined rules. In the case of the control system framework, one can configure the controller to record the various measurements for data analysis or to discard this option due to performance issues.
- *Specialization*: Specialization means that application elements are derived from the framework elements by extending and/or partially modifying the more general one. In the case of the control system framework, one can specialize a sensor into remote sensor, touch sensor, etc. Thus, these can be further used within the specific application.
- *Instantiation*: Instantiation means the selection of specific values for elements within the framework for specific applications. In the case of the control system framework an object of type controlled value can be instantiated. For example in the climate control system such instantiation include the element to be controlled (e.g., room) and the thresholds for signaling.
- *Realization*: Realization means the implementation of non-implemented hooks within a framework in the specific application. This mainly refers to abstract concepts. For example, in the control system framework a controller may be an abstract object that requires implementation as different applications require different control flows.

- *Use*: Use means that one should adopt the code as is. That means that no change to the code should be done. It mainly refers to the internals of the frameworks that should not be taken care of by application developers. For example, the communication among the various components within the control system framework is hidden from the developers and should not be changed.

3 The JADE Framework

The JADE agent platform [27] is a widely used framework for developing MAS. It was developed during the last decade and is compliant with various agent standardization communities, such as FIPA, IEEE, and OMG. JADE is a complete middleware and a programming paradigm that supports the development of MAS. It consists of the services required by MAS such as communication, security, agent management, from the infrastructure point of view and a publicly available source code that enables the implementation of MAS, from the programming point of view.

In this section, we provide a general overview of the JADE framework source code via a partial model and demonstrate weaknesses in its code listing that might cause problems in using the framework. The partial JADE model in Fig. 1 demonstrates the main concepts within JADE that comprise MAS (with respect to its implementation). An agent within JADE executes various assigned behaviors that are responsible for agent functionality and agent communication with the environment (e.g., other agents). These include message passing and the scheduling and execution of multiple concurrent activities. An agent consists of many behaviors related to many types; several of these might be composite and include other behaviors. Note that each agent might be related to some ontologies and behaviors, as well as to ACL messages. Also, note that JADE code (classes) is used for two purposes: the first is to enable the proper execution of the agents and the second is to facilitate the creation of new applications through inheritance and composition. Thus, the code has two parts, one of which should not be manipulated by the application developers (although it might be stated as “public”), and the other part that should or may be changed by the developers. However, developers may not be aware of this classification, as this knowledge is partially specified within the code and guidance may also appear in other documentation. For example, usually the internal implementation part of the *Agent* class as appears in Listing 1 should not be overridden by the developers. Also, the *setup* method should be overridden as it should initialize the specific agent activities; in case the method is not implemented, the agent functionality will not be executed. Furthermore, it is not clear how developers should handle the *doWait* method. In the *Behaviour* class that appears in Listing 2, the methods *action* and *done* should be overridden as directed by the abstract keyword. However, note that, for example, in the *OneShotBehaviour* class (not shown here), the *done* method is already implemented and should not be handled by the developer, unless she wishes to change the semantics of the behavior. It is not yet clear how to handle the *onEnd* method. Furthermore, it is not stated

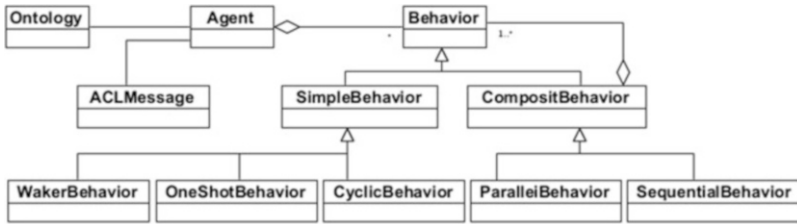


Fig. 1 A general and partial JADE model

```

public class Agent {
    // internal implementation
    ....
    ....
    protected void setup() {}
    protected void takeDown() {}
    public void doWait() {doWait(0);}
    ...
}

```

Listing 1 An excerpt of the agent class code within JADE

```

Public abstract class Behaviour implements Serializable {
    // internal implementation
    ....
    public abstract void action();
    public abstract boolean done();
    public int onEnd() { return 0;};
    public void onStart() {}
    ...
}

```

Listing 2 An excerpt of the behaviour class code within JADE

which classes are allowed to be composed or inherited and which classes can serve as a container for other classes. Furthermore, no specification of possible control flow is provided.

Having enumerated above the missing guidelines, we call for systematic guidance to enhance reusability. For this purpose, we adopted ADOM method, which is further elaborated in the next section.

4 The ADOM Approach

ADOM [25, 26] is a method that facilitates the specification of reusable assets and their reuse within specific application regardless of the specification language. ADOM supports the representation of domain models, construction of application-specific models, and validation of the application-specific models against the

relevant domain models. ADOM is rooted in the domain engineering discipline which is concerned with building reusable assets, on the one hand, and representing and managing knowledge in specific domains, on the other.

ADOM has three layers: (1) The language layer, (2) the domain layer, and (3) the application layer. The *language layer* comprises metamodels and specifications of the languages that are used to specify the domains and application models. ADOM can be implemented in any language; however, it requires a classification mechanism. That mechanism is used for specifying constraints (and guidance) within the domain layer and for connecting application elements to their corresponding domain elements. The *domain layer* holds the reusable 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, as well as guidance in how to reuse the elements within 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.

ADOM facilitates the specification of commonality, variability, and reusability by a set of indicators which are defined as follows. The *commonality specification* in ADOM is specified by the «multiplicity» indicator, which is used for specifying the range of the number of elements within an application, i.e., the application elements, which can be classified as the same domain element. Two tagged values, min and max, are used for defining the lowest and uppermost boundaries of that range.

For *variability specification*, ADOM provides two indicators. One is the «variation_point» indicator, which has the following tagged values: (1) open, specifying whether the variation point is open or closed, i.e., whether application-specific variants that are not specified in the domain can be added at this point or not, and (2) card, which stands for “cardinality,” indicating a variant’s selection rule in the form of the range of variant types needed to be chosen for this variation point. The second indicator is «variant», which can be a realization of a variation point and should be of the same type. A tagged value vp associated with the variant specifies the name of the corresponding variation point.

To allow for the specification of reusability guidance, ADOM uses the «reuse» indicator, which has the following tagged values: (1) *mechanism*, which can take different values representing the different applicable mechanisms, that is, configuration, aggregation, specialization, instantiation, realization, and use; (2) *base*, which determines whether the associated element can be used by the stated mechanism; for example, whether the element can be specialized, can compose other elements, can be configured, etc.; (3) *used*, which determines whether the element can be used

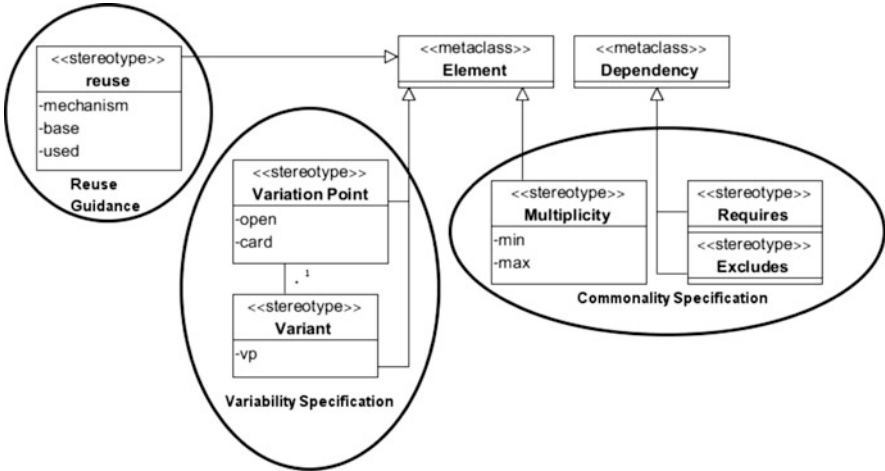


Fig. 2 A UML profile of ADOM

when applying the stated mechanism. For example, can it be composed, can it be passed as a parameter for configuration, etc.

In general, when an indicator in ADOM is associated with a domain element, its specifications are required to be fulfilled within the applications. If the indicator is not specified, then no constraints are imposed.

Summarizing ADOM indicators, Fig. 2 presents a UML profile of the various indicators—using the stereotypes classification mechanism built-in UML.

As stated before, the relations between a 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 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 a validation mechanism that prevents application developers from violating domain constraints and reusability guidelines while (re)using the domain elements 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.

To exemplify the ADOM method, consider a domain with one class called *Demo* that consists of two attributes *a* and *b* and is equipped with the following indicators: multiplicity: min = 1, max = ∞ ; variation_point: open = true; reuse: mechanism = aggregation, base = false, used = true. That means that applications

Table 1 Demonstrating the ADOM indicators

	Multiplicity	Variation_point	Reuse
<i>Domain specification</i>			
Demo	1,∞	open = true	mechanism = aggregation,
a	1,1		base = false, used = true
b	0,∞		
<i>Application specification</i>			
App1Class(Demo) x(a)	✓	✓	✓
App2Class(Demo) x y(b)	X	✓	✓
App3Class(Demo)	✓	✓	X
App4Class x(a)			

in the domain have to include at least one class classified as *Demo* and that class cannot aggregate other classes, yet, can be part of other classes. As it is also specified as an open variation point it can be extended by various application-specific variants.

Table 1 demonstrates the relationship between the domain layer and the application layer as well as the validation capabilities. Following the above example, the specification of the domain appears on the first three rows. The classification of the application elements is shown in brackets in the next three lines of the table. For the first case all constraints hold. For the second case, an attribute which is classified as “a” is missing. Since it is a mandatory attribute (see the multiplicity specification on the domain specification), it is a violation of the domain specifications. In the third case a violation occurs with respect to the reuse guidance as a “Demo” class aggregates another class—App4Class.

5 The ADOM-JAVA Dialect

In this chapter we deal with frameworks at the code level. Thus, we adopt ADOM along with Java as the underlying language. To fully apply the approach we use Java annotation¹ to satisfy the classification mechanism requirement, since it enables the specification of meta data. Listing 3 synthetically 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 having classes classified as *aDomainClass* at least *A* times and no more than *B* times. In the application layer the *aApplicationClass* class is classified by the *aDomainClass* class. Furthermore, the *aDomainClass* must consist of other classes and cannot be aggregated into other classes as indicated by the @reuse indicator. In addition, the *aDomainClass* cannot be specialized by the application classes and has to be used for configuration in one of the application classes. Note that no constraints exist regarding the configuration of the *aDomainClass*. In addition, in

¹In this work we use the Java Annotation called @Java [4].

```

// domain layer code
@multiplicity(min = A, max = B)
@reuse(mechanism = aggregation, base = true, used = false)
@reuse(mechanism = specialization, base = false)
@reuse(mechanism = configuration, used = true)
public class aDomainClass{}

// application layer code
@aDomainClass
public class aApplicationClass {
    appVar bApplicationClass;
    ...
}

```

Listing 3 Demonstrating the ADOM-JAVA syntax

```

//domain layer code
@multiplicity(min = 1)
@reuse(mechanism = specialization, base = true)
@reuse(mechanism = aggregation, base = true, used = false)
@reuse(mechanism = configuration, base = false, used = false)
public class Agent implements Runnable, Serializable {

    @multiplicity(min = 1, max = 1)
    @reuse(mechanism = specialization, base = false)
    @reuse(mechanism = aggregation, base = false, used = false)
    @reuse(mechanism = configuration, base = false, used = false)
    private class AssociationTB {...}

    @multiplicity(min = 1, max = 1)
    @reuse(mechanism = specialization, base = true)
    protected void setup() {}

    @multiplicity(min = 1, max = 1)
    public void doWait() {doWait(0);}
}

```

Listing 4 The JADE agent class with the ADOM indicators

the application layer code the *aClassApplication* class consists of the aggregation of the *bApplicationclass* via a reference variable; this is in line with the reuse specification, as it must consist of other classes. Nevertheless, even if this constraint was not specified, then the insertion of additional elements to the class is allowed and considered as application-specific elements.

As in this chapter we focus on reuse guidance, we emphasize the use of the reuse indicator and demonstrate it over the JADE code. Listing 4 presents a part of the original JADE Agent class along with the annotation suggested by ADOM-JAVA.

The annotations in the listing that appear before the class declaration have the following semantics (in the order they appear in the listing):

- There should be at least one agent class ($\text{min} = 1$).
- The agent class should be specialized.
- The agent class should be composed with other classes (or types) via data members but cannot participate in other containers.
- No configuration of the agent class is allowed.

```
//application layer code
@Agent
public class PingAgent extends Agent {

    private Logger myLogger =
        Logger.getMyLogger(getClass().getName());
    private class WaitPingAndReplyBehaviour extends
        CyclicBehaviour {.....}

    @setup
    protected void setup() {
        // Registration with the DF
        DFAgentDescription dfd = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("PingAgent");
        sd.setName(getName());
        sd.setOwnership("TILAB");
        dfd.setName(getAID());
        dfd.addServices(sd);
        try {...}
    }
}
```

Listing 5 The use of the agent class (adopted from the PingAgent example of JADE)

Next, the annotations for the private class *AssociationTB*, which is an internal implementation of the agent that refers to its management within the JADE framework, are presented. Since the class is intended to be internal, no changes or specializations to it are allowed. The annotations for the *setup* method state that there should be only one such method and it has to be specialized (by overriding it). Finally, the annotations for the *doWait* method state that there should be only one such method; however, its uses are not dictated (as there is already an implementation of that method).

Listing 5 presents an application that extends the class Agent of JADE. Following the ADOM approach, the application elements (class, attributes, and methods) are annotated with the framework element names. This facilitates the verification and enforcements of the constraints introduced within the framework. As can be seen, the PingAgent specializes the Agent class, and there are two application-specific elements that are allowed, as stated by the `@reuse(mechanism = aggregation, base = true)` statement. As the *setup* method has to be overridden, it also appears in that listing. The *doWait* method did not require any changes and is reused as is. This is also acceptable, as no constraint was specified.

To this end, we have shown the way that ADOM-JAVA supports the reusability guidance related to the structural nature of the application.

Nevertheless, ADOM-JAVA is applied to the behavioral aspect as well. In the following we present this notion. Listing 6 presents the implementation of the *checkInSequence* method of the *AchieveREInitiator* class within JADE. This class implements the FIPA-Request-like interaction protocols, in which the initiator sends a single message within the scope of a specific interaction protocol in order to verify if the RE (Rational Effect) of the communicative act has been achieved or not. The structure of such a protocol is as follows. The initiator sends a message, the responder can then reply by sending a *not-understood* or a *refuse* to achieve the

```

@multiplicity (min = 1, max =1)
protected boolean checkInSequence (@multiplicity (min = 1, max =1)
ACLMessage reply) {

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    String inReplyTo = reply.getInReplyTo();

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    Session s = (Session) sessions.get(inReplyTo);

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = aggregation, used = false)
    ifStat1: if (s != null) {
        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
        int perf = reply.getPerformative();

        @multiplicity (min = 1, max =1)
        @reuse (mechanism = use)
        ifStat2:if (s.update(perf)) {
            // The reply is compliant to the protocol
            @multiplicity (min = 1, max =1)
            @reuse (mechanism = use)
            switchStat1: switch (s.getState()) {

                @multiplicity (min = 1)
                @reuse (mechanism = realization, base = true, used =false)
                caseStat1: case @reuse (mechanism = instantiation)
                    Session.Status:
                @multiplicity (min = 1, max = 1)
                @reuse (mechanism = use, base = true, used =false)
                caseStat2: default:

                @multiplicity (min = 1, max = 1)
                @reuse (mechanism = use)
                failure1: return false;
            }
            // If the session is completed then remove it.
            @multiplicity (min = 1, max =1)
            @reuse (mechanism = use)
            ifStat3: if (s.isCompleted()) {
                @multiplicity (min = 1, max =1)
                @reuse (mechanism = use)
                remove:sessions.remove(inReplyTo);
            }
            @multiplicity (min = 1, max =1)
            @reuse (mechanism = use)
            success: return true;
        }
    }

    @multiplicity (min = 1, max =1)
    @reuse (mechanism = use)
    failure2: return false;
}

```

Listing 6 The implementation of the `checkInSequence` method within the `AchieveREInitiator` class—a generalized version

rational effect of the communicative act, or also an *agree* message to communicate the agreement to perform the communicative act.

The implementation presented in Listing 6 is adjusted from the JADE implementation and is equipped with the ADOM-JAVA indicators. Note that in this case,

```

@checkInSequence
protected boolean checkInSequence(@reply ACLMessage reply) {
    @inReplyTo String inReplyTo = reply.getInReplyTo();
    @s Session s = (Session) sessions.get(inReplyTo);
    @ifStat1 if (s != null) {
        @perf int perf = reply.getPerformative();
        @ifStat2 if (s.update(perf)) {
            @switchStat
            switch (s.getState()) {
                @caseStat1 case
                    Session.POSITIVE RESPONSE RECEIVED:
                @caseStat1 case
                    Session.NEGATIVE RESPONSE RECEIVED:
                    // The reply is a response
                    Vector allRsp = (Vector)
                        getDataStore().get(ALL_RESPONSES_KEY);
                    allRsp.addElement(reply);
                    break;
                @caseStat1 case
                    Session.RESULT_NOTIFICATION_RECEIVED:
                    // The reply is a resultNotification
                    Vector allNotif = (Vector)
                        getDataStore().get(ALL_RESULT_NOTIFICATIONS_KEY);
                    allNotif.addElement(reply);
                    break;
                @caseStat2 default:
                    @failure1 return false;
                    @ifStat3 if (s.isCompleted()) {
                        @remove sessions.remove(inReplyTo);
                    }
                    @success return true;
                }
            }
            @failure2: return false;
        }
    }
}

```

Listing 7 The implementation of the `checkInSequence` method within the `AchieveREInitiator` class—an implemented version

since the statements have no classifiers, we used the notion of labels to refer to these statements.

In this listing the following constraints should be imposed. There should be only one *checkInSequence* method with one parameter. The *inReply* variable should be used as is. The same holds for the *s* (of type *Session*). Next, the *ifStat1* should appear once in any application but may consist of additional element (i.e., statements), yet it cannot be composed into other blocks (used = false). Following the *perf*, the *ifStat2*, the *switchStat*, should appear only once and should be used as is. The *caseStat1* refers to the cases within the switch statement. In this case there might be several cases and these should be realized. Next, the other specifications of the domain elements are similar in that they should appear only once and used as is.

Listing 7 presents the application code of the `checkInSequence` method. The domain classification also uses Java annotation. The code can run without these annotations; yet, these are used for checking the compliance with respect to the guidelines provided within the framework code.

6 Other Applications of ADOM-JAVA

The idea of easing application development applies not only to the utilization of frameworks but to general development processes as well. Many efforts have been made in order to increase programming productivity (by reducing the development efforts) [19, 20]. For example, according to Jones [16], the development of third generation languages, which raised the level of abstraction and hide complexity, improved programming languages' productivity by 400 % compared to assembly (measured by the average number of source statements per function points). Furthermore, the object-oriented paradigm, which supports reusability through inheritance, improved programming productivity yet again. For instance, Java improved the productivity of Basic (a structural programming language) by an additional 20 % [16]. As stated in the introduction, many reuse techniques have been evolved, yet, these are general techniques and do not refer to specific domains. Current belief among the software engineering community advocates that future productivity improvements will only be achieved through utilization of commonalities in different domains [6, 7, 18, 28]. In order to assimilate this notion, domain-specific languages (DSLs) were devised [18, 21]. In general, DSLs are divided into two distinct types, external and internal DSLs, which we discuss next.

The basic premise of external DSLs is that the underlying principles of higher abstraction levels and tailoring to specific domains necessitate the development of the DSL from scratch. There is typically a domain expert with expertise in the semantics of the domain and an expert programmer with expertise in developing complicated and sophisticated software, both working on this process [18]. This process can be further divided into two sub-processes: design and implementation. The design process includes defining domain constructs 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 consists of the DSL's supporting tools. The code generator takes the DSL specifications as input and validates them according to the domain constraints, issues error reports if necessary and finally, transforms them to low level source code as output, optionally using a domain-specific framework for this transformation. As these tools were built by experts, the resultant code reuses domain and programming expertise. The main two advantages of external DSLs are improved productivity (mainly due to abstraction) and reuse of expert knowledge. However, external DSLs still suffer from various limitations. The design and implementation of external DSLs is complicated and time consuming. Even if the work is done by experts and supporting tools are available, it might not be enough to ensure a successful working DSL. According to [12], most DSL projects are usually abandoned in the development process and the work is eventually done in regular general purpose languages. Moreover, introducing the notion of DSL-based development into an organization requires significant changes in the organization's development paradigm. These changes require both new tools and new processes.

While some managers might be able to see the long-term advantages of DSLs, others might be reluctant to introduce radical, expensive, and time-consuming changes to their natural development process. All of these reasons indicate that the applicability of external DSLs is limited. Another limitation of external DSLs is their limited expressiveness. External DSLs confine the application developer to a pre-formulated set of the language constructs. As the language is tailored to a specific domain, it cannot be used to express semantics outside of the language's boundaries, which could have been wrongfully designed.

Internal DSLs draw their inspiration from the recognized drawbacks of external DSLs. Their basic premise is that DSLs should not be developed from scratch but rather they should be embedded in existing proven general purposed programming languages (GPPLs). In this sense, internal DSLs are no different than regular domain-specific application programming interfaces (APIs). However, they are different in the sense that the APIs are designed to resemble natural languages. This is achieved by advanced coding techniques such as method chaining, expression builders, interface chaining, and generics. [11]. The main advantages of internal DSLs is that they do not suffer from the above-mentioned drawbacks of external DSLs. This improvement results from three main reasons: (1) the development of internal DSLs is much easier with respect to external DSLs, mainly because the GPPL facilities (i.e., advanced IDEs) 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 processes; and (3) internal DSLs do not limit application developers' expressiveness as they allow the use of GPPL regularly. These reasons indicate that internal DSLs are more applicable than external DSLs. However, internal DSLs introduce the following limitations. External DSLs achieve improved code quality through pre-code generating validation algorithms and higher abstraction levels. Current reports of internal DSLs focus on code readability and maintainability [17]. Although these should have positive effects over productivity, it is hard to see how sophisticated APIs raise the level of abstraction similar to external DSLs. Also, although internal DSLs can exploit coding techniques in order to assure some domain semantics, they cannot implement validation algorithms that examine the specified code according to domain constraints. Ultimately, the application programmer's expressiveness is unconfined, thus she can use (or abuse) the API in any way, and therefore, internal DSLs are less productive than external DSLs.

To bridge the gaps between internal and external DSLs, we also apply ADOM-JAVA to further guide the developers, reducing their development efforts and thus increase their productivity and code quality. As we use two levels of abstraction, we gain the advantages of external DSLs and integrate the approach with GPPL we therefore gain the benefits of internal DSLs. Furthermore, the application of ADOM-JAVA may also be used for dictating architectural and programming styles.

7 Evaluation

In order to evaluate the use of ADOM-JAVA, we performed an experiment with undergraduate students to check whether the approach can lead to better code quality and to development time reduction. The research questions were the following: (1) Does the development of an application with ADOM-JAVA lead to better code quality with respect to the development of application with Java? (2) Does the development of application with ADOM-JAVA better address the functional requirements with respect to the development of application with Java? (3) Does the development of an application with ADOM-JAVA lead to a reduction of the development time with respect to the development of application with Java?

The subjects in the experiment were 50 undergraduate students in an Information Systems Engineering program at the Ben-Gurion University of the Negev, who participate in an “Object-Oriented Analysis and Design” course. During the course, the students studied ADOM and its capabilities and, in particular, the application of ADOM-JAVA. The study took place at the end of the course as a class assignment. The experiment consisted of two groups. The students in both groups received a requirement document of the application (a lab management system) and the group that uses ADOM-JAVA received the domain code as well as the supported tool. Note that the domain was familiar to all students, as it was part of the course material. The experiment lasted 9 hours in which the students had to provide a working application, which fits the requirements. The experiment took place in various sessions, where each session consists of students programming with regular Java and students programming with ADOM-JAVA. Their work was supervised to ensure that each student perform the task independently of the others. During the experiment the students were allowed to take breaks as the experiment duration was long. To encourage the students’ performance, we motivated them by adding a bonus to their final grades, in accordance with their achievements.

To check the outcome of the students, we measured the development time (hours), ran a series of tests to verify the functionality (number of tests passed), and examined the code structure in terms of layer separation and object responsibility assignments. Table 2 presents the experiment results. It can be seen that the students who used ADOM-JAVA achieved better results in all categories. This was very clear with respect to the development time as the differences were statistically significant (using Wilcoxon rank sum bidirectional test) and with respect to code quality. The differences related to the functionality were a bit lower (and not statistically significant) since the number of tests was low and the application was relatively simple.

In addition to the empirical analysis we interviewed the subjects who used the ADOM-JAVA. They mentioned that the approach indeed introduces development guidance, yet further training and tool improvement is required.

Although further examination is required, following the results it can be observed that although explicit domain knowledge may be cumbersome, it does have a positive effect over the development process.

Table 2 The experiment results

		ADOM-JAVA	Regular Java	Sig.
# Participants		26	24	
Development time (hours)	Average	7.49	8.43	0.001
	Std	1.22	0.93	
# of pass tests	Average	2	1.54	0.107
	Std	1.02	0.98	
# of problems concerning layer separation		0	8	
# of problems concerning responsibility assignments		5	16	

8 Summary

Reusability has long been discussed and addressed by both practitioners and researchers. The main goal of reusability is to increase productivity in terms of reduced development time and increased code quality. Many reuse techniques have been devised, yet their usage might introduce difficulties. In this chapter we refer to one of the common reuse techniques, namely, frameworks. It is well known that frameworks provide a rich knowledge and implementation for the application that uses them. Yet, due to the amount of knowledge (and implementation) encapsulated in these frameworks, it is difficult to apply these effectively. To bridge this gap, in this chapter we adopt a domain engineering approach, ADOM, adapt it to the context of programming and frameworks, and demonstrate how that approach guides the reusability of a given framework. We also evaluate the approach and find it useful for the task at hand.

While ADOM-JAVA looks promising in increasing developers' productivity with respect to frameworks and domain-specific languages, it is clear that additional examination is required. Currently, the meta model of ADOM consists of only the reuse mechanism with two type of constraints. Thus, we plan to examine the specification of the reuse indicators to further facilitate the reuse guidance. In particular, we are interested in examining which other reuse mechanisms are required to be supported and what are the parameters needed for their configuration. In addition, the usage of ADOM-Java should be further explored and evaluated.

References

1. Ali, Z., Bolinger, J., Herold, M., Lynch, T., Ramanathan, J., Ramnath, R.: Teaching object-oriented software design within the context of software frameworks. In: *Frontiers in Education Conference (FIE)*, pp. S3G-1–S3G-5. IEEE, Washington, DC (2011)
2. Becker, J., Janiesch, C., Pfeiffer, D.: Reuse mechanisms in situational method engineering. In: Ralyte, J., Brinkkemper, S., Henderson-Sellers, B. (eds.) *Situational Method Engineering: Fundamentals and Experiences*. Springer, Boston (2007)
3. vom Brocke, J.: Design principles for reference modelling—reusing information models by means of aggregation, specialisation, instantiation, and analogy. In: Fettke, L. (ed.)

- Reference Modeling for Business Systems Analysis, pp. 47–75. Idea Group Publishing, Hershey (2007)
4. Cazzola, W.: @Java: A Java Annotation extension. <http://cazzola.di.unimi.it/atjava.html>. Last accessed April 2013
 5. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, Boston (2001)
 6. Czarnecki, K., Eisenecker, U.W.: Generative Programming—Methods, Tools, and Applications. Addison-Wesley, Boston (2000)
 7. Czarnecki, K.: Overview of generative software development. In: Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, France (2004)
 8. Fayad, M., Schmidt, D.C.: Object-oriented application frameworks. *Commun. ACM* **40**(10), 32–38 (1997)
 9. Fayad, M., Schmidt, D.C., Johnson, R.: Implementing Application Frameworks: Object-Oriented Frameworks at Work, 1st edn. Wiley, New York (1999)
 10. Fayad, M., Johnson, R.: Domain-Specific Application Frameworks. Wiley, New York (2000)
 11. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Boston (2010)
 12. Freeman, S., 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, pp. 855–865. ACM, New York (2006)
 13. Harsu, M.: A survey on domain engineering. Report 31, Institute of Software Systems, Tampere University of Technology (2002)
 14. ISO 9126.: ISO/IEC TR 9126-software engineering, product quality, quality model. International Organization for Standardization, Geneva (2001)
 15. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM, New York (1997)
 16. Jones, C.: Estimating Software Costs. McGraw-Hill, New York (2007)
 17. Kabanov, J., Raudjärv, R.: Embedded typesafe domain specific languages for Java. In: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp 189–197. ACM, New York (2008)
 18. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, New Jersey (2008)
 19. Kiebertz, R.B., McKinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L.: A software engineering experiment in software component generation. In: Proceedings of the 18th International Conference on Software Engineering, pp. 542–552. IEEE Computer Society, Washington, DC (1996)
 20. Lowell, A.J.: Programmer Productivity: Myths, Methods, and Morphology. A Guide for Managers, Analysts, and Programmers. Wiley, New York (1983)
 21. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv.* **37**(4), 316–344 (2005)
 22. Polancic, G., Hericko, M., Pavlic, L.: An empirical examination of application frameworks success based on technology acceptance model. *J. Syst. Softw.* **83**, 574–584 (2010)
 23. Polancic, G., Hericko, M., Pavlic, L.: Developers’ perceptions of object-oriented frameworks—an investigation into the impact of technological and individual characteristics. *Comput. Hum. Behav.* **27**, 730–740 (2011)
 24. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, New York (2005)
 25. Reinhartz-Berger, I., Sturm, A.: Enhancing UML models: a domain analysis approach. *J. Database. Manag.* **19**(1), 74–94 (2008). Special issue on UML Topics
 26. Reinhartz-Berger, I., Sturm, A.: Utilizing domain models for application design and validation. *Info. Software. Technol.* **51**(8), 1275–1289 (2009)
 27. Tilab.: JADE—Java Agent DEvelopment Framework. <http://jade.tilab.com/index.html>. Last accessed April 2013
 28. Weiss, D.M., Tau, C., Lai, R.: Software Product Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Boston (1999)

Domain Engineering

Product Lines, Languages, and Conceptual Models

Reinhartz-Berger, I.; Sturm, A.; Clark, T.; Cohen, S.;

Bettin, J. (Eds.)

2013, XVI, 404 p., Hardcover

ISBN: 978-3-642-36653-6