Implementing a Domain-Driven Design in Java

Ingmar van der Steen

February 8, 2019

1 What is a domain-driven design?

As a developer you are confronted with a lot of technical problems, but automating the business in the way your client asks for is even more challenging. In order to bring business and IT closer we adopt domain-driven design (DDD). This way of designing software helps in communicating your software to less technical domain experts and have focus on the problem domain itself rather than only fixing technical problems to implement the software. It is a way of designing software that is characterized by having a strong link between a model of the domain for which software is developed and the actual implementation of this software.

A model is an abstraction of reality and hides extraneous details about it not relevant to the problem at hand. The domain and its business rules take center stage in the development process which is clearly reflected in the code. Besides this strong link between model and implementation, there needs to be good communication between developers and domain experts to iteratively refine this model throughout the project.

This article will study the applicability of DDD in the domain of an on-line banking application which we call Payments of which the source can be found here:[2]. The study is based on the concepts and ideas introduced by Evans[1] and are applied in the Payments application. In this application a customer can add a payment to the system, and edit a payment by changing its attributes. The study will lead to the answers of the following questions:

1. Why would we use a domain-driven design in our CRUD application?
2. How do we apply domain-driven design in terms of communication, design and process?
3. Which trade-offs are made in such an application when this design is adopted?

In DDD the primary focus needs to be on the domain and its rules and not the technical implementation details. Often the link between the model of the domain and the implementation is cluttered by this technical noise. Although technical problems are solved the identification of the domain in our codebase

is lost. Resulting in a gap between the domain model and its implementation. Communication between domain experts and developers is difficult since they now find two different models of the domain in which requirements are hard to trace.

On their turn, domain experts need to commit more on paper in such a way that it translates easily into the domain model and developers can implement it. This process will also clarify business rules and make them explicit, solve potential contradicting rules and place other rules out of scope. Since developers and domain experts will speak the same language in terms of the domain model, there is a smaller gap between the model and its implementation and therefore a smaller gap between domain experts and developers. This results in early adoption of the software and can be used as direct feedback to the domain model implemented. A strong link between model and implementation makes it much more explainable and transparent where problems exist in the model and where we need to improve.

Moreover, the reason why we want to have a domain-driven design is that we can easily trace the business rules in the code base. This way it serves as the answer to the coverage of requirements and we can easily identify if requirements that are not committed on paper are present in the application. In a dynamically sized team new team members will be able to understand the domain more easily because it is more present in the code. If new business requirements are to be implemented it will be much more clear where and how we need to implement this.

2 The Payments domain

The domain of the Payments application was already briefly mentioned, it involves making payments from one customer to another. The following definitions describe the most important concepts in our domain:

DEF 2.1 *Payment Instruction: models a payment that is made from an Ordering Account to a Beneficiary Account.*

The Payment Instruction has a couple of relations of which the following are the most important:

DEF 2.2 *Ordering Account: the account of the payment which belongs to the bank for which this Payment System is made. The Ordering Account is the account from which the money is debited. It is in the IBAN format (International Bank Account Number)*

DEF 2.3 *Beneficiary Account: the beneficiary is the party to which the money is credited. The account is composed of an account identification and name. The account identification is in the IBAN format.*

Besides the Ordering Account and Beneficiary Account, we need to supply additional information in the Payment Instruction:

DEF 2.4 *Currency: the currency of the payment (e.g.: USD, EUR)*

DEF 2.5 *Amount: the amount of money which needs to be transferred from the Ordering Account to the Beneficiary Account in the Currency given for the Payment Instruction.*

DEF 2.6 *Forward Date: date on which the payment is scheduled to be send through the banking network*

DEF 2.7 *Fedwire Code: a code for banks used in the United States which identifies financial institutions. Fedwire codes are always 9 digits in length.*

DEF 2.8 *Manual Bank: a bank described by a name, address and country which is not known by the Payment system and therefore not validated on its existence. In contrast to a bank identified by a Bank Identification Code, i.e. BIC.*

DEF 2.9 *Beneficiary Bank: the bank to which the Beneficiary Account belongs. The Beneficiary Bank is a Manual Bank.*

2.1 Communicating the domain

Developers and domain experts need to communicate about the domain using an ubiquitous language[1]. If there are meetings set up between them discussing requirements of the system, they need to speak the same language in terms of domain concepts and the relations between them. If this is not the case there will be information lost in translation or people in fact do not understand each other and consequently wrong decisions are made. When developers do not tightly link the domain concepts agreed upon to the code, they tend not to use these concepts intensively in discussions with the domain experts.

Moreover, they will use concepts from their technical design which the experts do not understand directly and need to be translated to concepts that make sense to the expert. This translation is time consuming and we have the risk that due to miscommunication the requirements are not implemented correctly.

When all members of a team with different disciplines use this language, all documentation written for the system become much more usable and traceable. It is often the case that a lot of knowledge exist in the minds of experts which are not trusted to paper. Moreover, they use jargon which adds another layer of translation. If we use an ubiquitous language and the knowledge is written down in this language, the utility of this documentation becomes much higher for future reference. The language should be used on a fine-grained level. This implies that we need to be consistent with naming of classes, members and operations which model the concepts. A change in a definition of a term in the language should result in a change of behavior in the code. Team members need to train themselves to use an ubiquitous language and the responsibility lies in the team to correct each other in this matter. This involves correcting it in the code base, requirements, SCRUM artifacts and in day-to-day meetings.

In the worst case people could assume you mean the same thing as they do, but this need not be true. For instance if we talk about the requirement that a user should be able to add a Payment Instruction which targets an Ordering Account and a beneficiary we all need to have the same understanding of these business concepts.

The use of this ubiquitous language aids in the adoption of Behavior-Driven Development[3] in a project. In BDD we specify requirements in terms of behavior of the software. If this specification is written in this language using an agreed upon structure as shown below, the specification is clear to everybody in the project. Subsequently, we can more easily implement unit tests based on the specifications written by domain experts. We know exactly what to test, how much to test and what to call the test. Moreover, since we are using a standard structure it is easy to automatically generate skeleton code for these tests (e.g.:[4]). The following will demonstrate using BDD, how we can specify validation rules on bank account numbers which are applied when validating the Payment Instruction's Beneficiary Account:

Story	IBAN validations on the Beneficiary Account of a payment
In order to	add a valid Payment Instruction
As a	user of the payments module
I want to	receive a validation error when the Payment Instruction is not valid

Scenario 1	IBAN checksum
Given	a Beneficiary Account
And	the account does not pass the IBAN checksum
When	the user add the Payment Instruction
Then	I should receive a validation error that the given Beneficiary Account is not a valid IBAN account

As we can see in the above example we are using terms of our ubiquitous language, e.g.: Payment Instruction and a Beneficiary Account. These terms fit into a template with a clear Given,When,Then set-up that comes naturally when writing down specifications. Scenario 1 translates easily into a single unit test with clear preconditions (Given,And), a trigger (When) and the expected outcome (Then). Moreover, in the header of this specification it becomes clear which business value it has and which stakeholder is involved.

3 Binding the domain and implementation

As stated before there needs to be a tight coupling between the domain model and implementation. It is observed that domain experts tend to create some upfront analysis model of the domain which is abandoned when implementation begins. The disadvantage of this is that the effort is lost when creating the design for the implementation. In the worst case problems and solutions are

rediscovered in the phase of implementation and potentially lead to differences between the analysis model and implementation.

When we look at the Object Oriented paradigm, there is no need to let the analysis model and implementation of the model differ greatly. Conceptual objects can naturally be expressed in an object oriented language like Java. The idea is that business and IT look at the same model which can directly exists in code instead of having two separate ones. However, naturally expressing the model can naively violate software design principles. As a result you will always tweak the model in order to use a design pattern. But these technical considerations should not weaken the model in such a way that it is easy to implement but the connection with the domain is lost. It is therefore no surprise that there is a feedback cycle between the model and design that serves for balancing between these two that are connected by the OO paradigm

In order to let the implementation express the model effectively we need particular decisions and code constructs to do so. For this we introduce a pattern language that describe these constructs and are covered in the next sections. Referring to the latter picture, this language will bind the domain model to our design using the OO paradigm.

3.1 Isolating the domain with a layered architecture

When our goal is to effectively express the domain model in our implementation we first need to make sure we are isolating the domain from technical details supporting that domain for e.g. persistence in the database. Otherwise, the binding between the two becomes cluttered with these details. For most developers it comes second nature to make a layered application in which a layer on top of another makes public API calls to the one beneath it. This layering serves mainly the purpose of high cohesion and understandability of this layer and being able to distribute these layers across different environments. Imperative to domain-driven design is that we isolate the domain on its own layer which consists of domain concepts and accompanied business rules.

A common mistake is introducing the 'Fat service layer'[1] in which a lot of domain functionality is implemented and mixed with technical supporting code. We need to isolate the domain layer from the service layer well enough and the latter should only define what the software is supposed to do and asks the rich domain objects to work out problems. Therefore, the service layer should not contain business rules or knowledge avoiding a fat service layer. The advantage of doing so is that we don't scatter or duplicate the domain across layers and it promotes reuse of the domain layer in another type of application. When a business rule changes we do not have to scan all layers of the application on which we need to make this change.

The domain objects do not need to concern themselves how they are displayed in the UI, nor how they are stored in the database. These actions are the responsibility of the UI and infrastructure layer respectively. Therefore, there must not be noise from direct dependencies on an ORM framework or a serialization framework in these domain objects. In the rest of this section it is

demonstrated how we isolate the domain layer using an example.

In the Payments application we can execute CRUD operations on a Payment Instruction(PI for short). The following two operations were already briefly mentioned:

1. Add a new Payment Instruction
2. Edit an existing Payment Instruction

When a PI is added or edited it is validated and subsequently stored in the database. These validations prevent the user to enter payments in the system which contain invalid data. An example of a validation is the following written using a BDD template:

```
IF the Beneficiary Bank is entered (1)
AND the Country of the Beneficiary Bank = USA (2)
AND the Currency = USD (3)
THEN the Fedwire Code is mandatory
```

The pages that support the new/edit operations are dynamic in behavior. An example of such behavior is that we only want to display the Fedwire input field when the above three preconditions hold. This implies we have the same business rule that drives front-end behavior and the mandatory check. We need to place the business rule of the Fedwire Code in the domain layer. In this way we can easily trace requirements in our code base and make this explicit to domain experts. However, this same business rule drives the web page behavior of creating a new Payment Instruction. When (1), (2) and (3) hold the input field needs to be made visible in the UI.

Since we do not want to duplicate the domain across layers we need to avoid having an UI layer incorporating this same rule only for visbility purposes. Otherwise, this so called 'smart' UI can quickly contain duplication of the business rules.

The following sections describe how the Payment's front-end looks like when we adopt a layered architecture that isolates the domain from all the other layers. The running example consists of a dynamic web page for adding a new Payment Instruction by posting its fields and the Payment Instruction is validated using the business rules.

3.2 Modelling business rules

Our goal is to isolate the domain and business rules from other layers and reusing this layer for the behavior of the UI. For the validation of a PI we define a business rule that checks if the Fedwire Code is mandatory given the three preconditions stated earlier. These kind of business rules can quickly become complex pieces of code for which the implementation could depend on other domain objects and resources. Therefore, we isolate these rules even further inside the domain layer. We will not put these rules inside the PI domain object, because this can quickly clutter the meaning and understanding of a PI itself. These rules are modelled as objects themselves using the SemanticBusinessRule

interface and take a domain object as argument and throw a BusinessRuleNotSatisfied exception holding the validation messages. Moreover, the isApplicable method models the precondition for the rule.

```
public interface SemanticBusinessRule {

default boolean isApplicable(List<String> params) { return true; }

void satisfiedBy(PaymentInstruction pi) throws BusinessRuleNotSatisfied;
}
```

The FedwireMandatoryRule implements the SemanticBusinessRule interface based on the BDD template above. It depends on the predicates isUSD and isUSA of our domain which belong to the domain objects Currency and Country respectively. Isolating the domain and business rules in a layer in this way is the first step to our goal. The second step is how we can share this implemented rule in the UI to implement the dynamic behavior. Namely, make the Fedwire Code field visible in our UI when the precondition holds. The challenge is to check the isApplicable() precondition on the back-end each time one or more of its dependencies (Beneficiary Bank country and Currency) are changed.

We can implement sharing business rules between UI and back end using computed observables in the binding framework KnockoutJS[5]. A generic isVisible function is introduced that executes an ajax call which contains a qualifier of the business rule and the values it depends on (i.e.: country and Currency). This function returns a boolean whether the field under consideration needs to be visible or not.

```
_this.fedwireFieldVisible = ko.observable();

_this.isVisible = function isVisible(rule, params, callback) {
   $.ajax({
               async: true,
               type: 'GET',
               data: {rule : rule, params: params.toString()},
               url: '${pageContext.request.contextPath}/isVisible',
               dataType: 'json',
               success: function (data, textStatus, jqXHR) {
                callback(data);
               },
               error: function (jqXHR, textStatus, errorThrown) {
                   alert(errorThrown);
               }
           });
};

_this.computeFedwireFieldVisibility = ko.computed(function() {
   _this.isVisible(
         "FEDWIRE_MANDATORY_RULE",
         [_this.selectedCurrency(), _this.selectedBankCountry()],
        _this.fedwireFieldVisible);

}).extend({rateLimit: 0});
```

Consequently, a visible binding is used in the HTML to decide whether we need to render the field or not:

```
<tr data-bind="visible:fedwireFieldVisible">
 <td>
  <input type="text" name="fedwireCode" data-bind="value:fedwireCode">
 </td>
</tr>
```

We have some overhead of doing an ajax call each time an user changes a dependency. Therefore, it is decided we do not serialize and post the whole form when calculating visibility of a field. Instead, only its dependencies are posted and the call is made asynchronous since we depend on back-end logic which otherwise could end-up freezing your UI. The disadvantage of this approach is that we need to make explicit these dependencies of the business rule in the JavaScript code instead of letting the back-end decide which dependencies apply to a particular rule. Note that the isVisible function is called two times when we e.g. edit a PI and restoring the data in the observables of the form: once for the Currency data and the second time for the country. In order to make only one call in this scenario, we introduce a debounced computed observable using rateLimits[5]. In this way we have less network traffic and no race conditions.

We have now a way to share and isolate our business rule on the domain layer. This isolation prevents implementing the smart UI anti-pattern in which business logic is put into the UI layer. The Fedwire business rule can easily be traced in our JavaScript and Java code following the FEDWIRE_MANDATORY_RULE qualifier. If the rule changes we only need to change it in one place and can be communicated to domain experts by just showing the code. The latter is an advantage since IT and the business can speak about the same model.

Because of separating the UI layer from the domain layer, there is a mapping defined from a PaymentInstructionDto to a PaymentInstruction domain object. In later sections we will also see that the persisted PI entity contains ORM meta data that should not belong in a domain object. Consequently, there will also be a mapping from the domain object to an entity that can be persisted into a database. In this way domain experts don't need to worry about technical implementation details of how this object is being persisted. Also we force a developer to think about the domain more thoroughly and to model it in code in a more natural way the business looks at it. The advantage of this approach is that the model directly lives in code and is identifiable and maintained by the developer and the business. The entity is allowed to have a different shape for e.g. optimization reasons, but needs to be isolated on its own layer. This section covered the way of how to isolate the domain model in a layer. A more detailed description of how to express the domain model inside the domain layer is given in the next section.

4 Introducing the pattern language

Evans[1] identifies five concepts in its pattern language which are used to express the domain model: Entities, Value Objects, Associations, Domain Services and Modules. These concepts are the building blocks of the domain layer and are defined in the following sections.

4.1 Entities

DEF 4.1 *Entity: objects which have continuity through a life cycle and are being distinct independent of its attribute values over time.*

This implies that state can change of such an object through time but its identifier stays immutable throughout. On the domain layer we can immediately identify such an entity: the PaymentInstruction. Its attributes can change over time by, e.g. editing the PI. However, it needs to have an identifier which should stay the same. We could have two PI objects which are equal in their attributes, but are in fact two complete different Payment Instructions. There is no natural key which makes a PI unique and therefore we need to recognize a paymentInstructionID. It is important to express this identity in the domain object since the business is concerned with the capability of processing two PI's that are equal in attributes as two individual instructions. Expressing this property by implementing the hasSameIdentityAs method from the introduced Entity interface in our PI object clearly models this intention. This method only compares the paymentInstructionID's and disregards all other attributes:

```
public interface Entity<T> {
   boolean hasSameIdentityAs(T entity);
}
```

There is no override of Object.equals used to express this property since such a method could have different understandings given the generic method name. One interpretation is that if equals() returns true on two objects, we are free to replace one instance for the other. However, in our case only comparing the ID in an implementation of equals will be dangerous since an entity can have the same ID but different values for its attributes over time and therefore make an incorrect replacement.

As described in the Java API[6], methods in the Collections Framework rely on the equals method of the data structure that is added or removed. It could be useful that we want to manipulate all the historic versions of the Payment Instruction's state in a collection and therefore need to implement an equals which complies to the above stated interpretation. If we compare all attributes of the PI (including the ID) we know that one instance can be replaced for the other if this method returns true. Consequently, this allows for e.g. adding PI's to a Set implementation:

DEF 4.2 *A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2)*

If we were to implement the equals method then we need to override hashCode as well, since in its contract is stated:

DEF 4.3 *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*

When we implement the equals/hashCode methods we add some functionality that does not relate directly to our domain. Evans[1] does not make any statement about equals nor hashCode but it would be naive not to implement them when our domain objects live inside collections. Therefore, we sacrifice a little bit of the separation of domain and technical details.

4.2 Value Objects

Besides having objects with identity we also create Value Objects which have no conceptual identity.

DEF 4.4 *Value Objects: objects that represent a descriptive aspect of the domain with no conceptual identity.*

Classifying an object as being a Value Object is essentially saying we only care about the values of its attributes and we don't have to identify it through time. Examples of Value Objects in our domain are: ManualBeneficiaryBank, Currency and Country. As an example, the first object is composed from name, address and country and are free format to the user. We do not even know if the supplied values in these attributes are valid and can potentially be subject to manual post processing of the PI. When these attributes change over time we are actually dealing with another supplied Manual Bank and no identity needs to be preserved for the application. Since Value Objects do not have identity we only have to implement the equals/hashCode pair and no implementation of hasSameIdentityAs exists.

4.3 Associations

Defining associations between objects is expressing links between those objects. We can define these associations in an uni- or bi-directional way. Deciding on how we can traverse these objects, boils down to answering the question if it is important to talk about e.g. a Currency and knowing to which PaymentInstruction it belongs. In our system we do not need to traverse from this Value Object to its associated entity object. It makes more sense to start from a PaymentInstruction and know which Currency it has. Therefore, we make this association uni-directional. The same holds for ManualBeneficiaryBank and Country.

4.4 Domain Services

Besides having Entities and Value Objects, a Domain Service is defined as:

DEF 4.5 *A process or transformation in the domain which is not a natural responsibility of an Entity or a Value Object.*

It is characterized by two properties:

1. The interface is defined in terms of other elements of the domain model.
2. The service is stateless.

The second property has the advantage that we always have predictable behavior given a domain element as input and forces the developer to consider whether we are really defining a service or an Entity or Value Object. An example of such a domain service is the ForwardDateService which calculates the Forward Date of a Payment Instruction based on its Currency. It is not natural to place this decision logic inside the PI entity itself since it adds noise to what a PI means. However, we introduce the forwardDate inside the PI because we want to express that a Payment Instruction has a date on which it gets forwarded. In the next sections we will see how we can manage this domain service in a Spring context without adding technical noise to the domain layer itself.

4.5 Modules

The last concept in our pattern language is Modules. We need modules/packages in our domain-driven design to group cohesive domain concepts together in order for a developer or business expert to communicate and identify more easily one particular part of a potentially large domain. As Evans stated, the meaning of the objects needs to drive the choice of Modules. Our domain lives under:

1. com.infosupport.poc.ddd.domain.entity.*
2. com.infosupport.poc.ddd.domain.rule.*
3. com.infosupport.poc.ddd.domain.service.*

The first package groups together the entity PaymentInstruction, the entity OrderingAccount and its associated Value Objects which form a cohesive group. The rules that are applied on the PI are grouped together into the rule package over which we can speak in isolation of all the details of the PI and its Value Objects. The last module we identify contains the Domain Services. In this module we put the ForwardDateService which set the Forward Date of the PI.

The packages:

1. com.infosupport.poc.ddd.dao
2. com.infosupport.poc.ddd.ui
3. com.infosupport.poc.ddd.service
4. com.infosupport.poc.ddd.service.dto

depend on these three domain packages and no dependencies exist the other way around. This means we can isolate the domain which can be potentially be reused in multiple projects without having dependencies on technical frameworks. Also note that the dao package contains a technical implementation of the PaymenInstruction and OrderingAccount entities in order to persist it in the database.

For banks, it is important to trace the payments which flow through the system. For auditing reasons we need a way to have output of what the system is doing. On the domain layer we want to have a trace when a payment is validated and which validation rule is applied. Since our domain layer needs to be kept clean from technical dependencies, we introduce an interface Tracer which abstracts from the way we are going to implement this trace. This interface is allowed to live on the domain layer, because for a domain expert it has meaning: the ability of tracing payments in the system.

An implementation would be that we log what the system is doing in the console. For this, we introduce a LogTracerImpl on the service layer which can be injected in an arbitrary object in the domain layer. In this way we let the service layer instantiate this technical implementation instead of the object itself in the domain layer. The object only holds a reference of type Trace to the tracer and allows for different ways of tracing (e.g. to a file, database etc.)

The adoption of this Separate Interface[7] pattern lets the domain layer be agnostic about the concrete implementation of the Trace interface. We separate the interface from the implementation by defining them in different packages. Since we are building a Spring MVC application we also want to benefit from references to domain objects being managed by the Spring application context. An example is the ForwardDateService introduced in the previous sections. However we do not want to violate the constraint that we introduce Spring dependencies into the domain itself. We can achieve this by using the annotation @Configuration and @Bean in a class called AppConfig outside the domain:

```
@Configuration
public class AppConfig {
    @Bean
    public ForwardDateService forwardDateService() {
        return new ForwardDateServiceImpl();
    }
}
```

This will promote the use of @Resource annotation in the places where we want to have this ForwarDateService being injected by Spring:

```
@Resource
private ForwardDateService forwardDateService;
```

Because of this independence on the Spring framework we can more easily share the domain layer with other teams and package it in a simple JAR file. When isolating the domain in a Maven CI this independence can be enforced build time using the maven-enforcer-plugin[8] to whitelist dependencies you allow in the domain.

5 Lifecycle of a domain object

The domain objects identified in the section of identifying Entities and Value Objects have associations with each other. For example, a PaymentInstruction holds an uni-directional link to BeneficiaryAccount. When we e.g. edit a PI it is important that we know which associated objects can be affected by a change and also which objects are directly exposed to the client. The former is even more important in a concurrent setting since we could end up changing an associated object simultaneously which can cause unpredictable outcomes.

5.1 Aggregates

To encapsulate an object and its associations, the concept of an Aggregate is introduced:

DEF 5.1 *Aggregate: a cluster of associated objects that is treated as a unit for the purpose of data changes.*

Such an Aggregate has a root and a boundary. The boundary defines what is inside the Aggregate and the root is a single Entity contained in this boundary. A client can only hold a reference to the root and all other objects in the boundary are in context of the root Entity. This implies we are not interested in those objects in isolation. The rules that apply to an Aggregate are the following:

1. Root Entities have global identity. Entities inside the boundary are only unique within the Aggregate (local identity).
2. Nothing outside the Aggregate boundary can hold a reference to anything inside, except to the root Entity.
3. Only Aggregate roots can be obtained directly with database queries. Other objects inside the boundary must be found by traversal of associations.

A clear Aggregate root in the Payments domain is the PaymentInstruction entity. It makes sense that a client can hold a reference to a PI. The associated objects: BeneficiaryAccount, ManualBeneficiaryBank are inside the Aggregate's boundary. As stated earlier we are not interested in holding a reference to only

a BeneficiaryAccount without an association with a PI. The Value Objects Currency and Country are outside the aggregate boundary since we need references to these objects in the semantic business rule for Fedwire:

```
public boolean isApplicable(Currency currency, Country country) {
   return currency.isUSD() && country.isUSA();
}
```

An Ordering Account is an Entity which does not have local identity, i.e. is not only unique within the Aggregate. It has global identity because it is relevant outside the Aggregate boundary of the PaymentInstruction. It could be the case that we want to do maintenance on an account, e.g. deactivate it or view all accounts details for a particular customer of the bank. Therefore, we define an Ordering Account itself as an Aggregate root, otherwise it conflicts with rule (1). This implies that an Ordering Account is not inside the Aggregate boundary any more of the Payment Instruction. Being a root Entity, the OrderingAccount implements the Entity interface as well. The implementation of the hasSameIdentity method is trivial by comparing the Ordering Account identification numbers.

With this encapsulation we can now operate on an Aggregate as a single unit in order to create, store, restore and modify it using Factories and Repositories that are defined in the following sections.

5.2 Factories

Aggregates can become complex object graphs of which the construction should not be the responsibility of these objects involved. Calling constructors sacrifices encapsulation of the Aggregates and it clouds the design of your objects. Factories create such object graphs for us and do not belong to the domain themselves. We introduce factories on the service and infra layer instead. The following snippet shows how to create a PaymenInstruction domain object using the PaymentInstructionFactory:

```
@Resource
private ForwardDateService forwardDateService;

public PaymentInstruction createPI(PaymentInstructionDTO dto) throws
    BusinessRuleNotSatisfied {

    List<String> msgs = new ArrayList<>();

    Currency ccy = Currency.create(dto.getPaymentCurrency(), msgs);
    Country cntry = Country.create(dto.getBeneficiaryBankCountry());

return new PaymentInstruction(
      OrderingAccount.create(dto.getAccountIdentification(), msgs),
      dto.getBeneficiaryAccountIdentification,
      ccy,
      forwardDateService.calculateForwardDate(ccy),
      new LogTracerImpl(PaymentInstruction.class),
      msgs);
}
```

This factory has as input the data transfer object PaymentInstructionDTO which contains the raw fields that was supplied by the user of the payment system. It calls the constructor of the PaymenInstruction domain object supplying parameters from this DTO. The entity OrderingAccount and the value objects Currency and Country are created by calling their static create() method, possibly returning validation messages when this object cannot be created.

The advantage of such a create() method is that we can return null with the reason why we did not return a reference to such a value object. Also it prevents in creating an invalid Domain Object according to our business rules. If you hold a non-null reference to such an object you are guaranteed to have a valid Domain Object which you can use throughout the application. Note that these Domain Objects are immutable and therefore we cannot mutate into invalid state which would also violate the TOCTOU principle[9]. The following snippet shows the implementation of the create() method for the value object Currency:

```
private Currency(String code) throws BusinessRuleNotSatisfied {
   if (code == null || code.length() != 3) {
          throw new BusinessRuleNotSatisfied("Invalid Currency");
  }
      this.currencyCode = code;
}

public static Currency create(String currency, List<String> msgs) {
     try {
          return new Currency(currency);
      } catch (final BusinessRuleNotSatisfied businessRuleNotSatisfied)
          {
          msgs.addAll(businessRuleNotSatisfied.getValidationMessages());
      }
      return null;
}
```

The Currency object is created when the raw input is a string of 3 characters and otherwise a validation message is added and null is returned. The parameter for the validation messages can be used to accumulate all messages that occurred during the creation of the PaymenInstruction entity and are returned to the user of the system.

Also in the case of the creation of a Fedwire object we now can model the dependency on Currency and Country in the create() method which forces the clients to also supply these other Domain parameters:

```
static Fedwire create(
 String fedwire, Currency ccy, Country cntry, List<String> msgs
) { }
```

In order to create a Fedwire obect we need to supply some Domain context as well in the form of a Currency and a Country. The create() method for a Fedwire is declared package private to enforce that only when we create the Aggregate root PaymentInstruction we can call the create() on the Fedwire class which lives in the same package as the PaymentInstruction, i.e.:

1. com.infosupport.poc.ddd.domain.entity.paymentinstruction

This models the dependence between the Aggregate root and the objects in its boundary. The create() for the Currency object is declared public since it is not part of the Aggregate boundary of the PaymentInstruction. An additional advantage of using factories is that we can call Domain Services which can do calculations, e.g. the Forward Date of a PaymentInstruction. This service is injected using the @Resource annotation and used in the factory. The mapping from an immutable Domain Object into a Data Transfer Object again is done using a bean-to-bean mapper[10]:

```
public PaymentInstructionDTO createDTO(PaymentInstruction pi) {
   return DozerBeanMapperSingletonWrapper
         .getInstance()
         .map(pi, PaymentInstructionDTO.class);
}
```

This mapper simply maps the data into the DTO using reflection and safes a lot of boilerplate code. We cannot use such a mapper if we are creating a Domain Object, because it will not give you validation when constructing the Aggregate.

5.3 Repositories

Subsequently, we need to store/retrieve the PaymentInstruction into/from the database. For this we introduce the Repository pattern that represents all objects of a certain type as a conceptual set. It acts like a collection, except with more elaborate querying capability. The machinery behind the Repository inserts or retrieves the PI's from the database. A Repository acts on the PaymentInstruction Aggregate using the PaymentInstructionRepository interface. This interface is defined on the domain layer and separated from the PaymentInstructionRepositoryImpl that is defined on the infra layer. Again, with a Repository we isolate the domain layer from technical implementation details of storing domain objects in the database:

```
public interface PaymentInstructionRepository {
   void add(PaymentInstruction paymentInstruction);

   PaymentInstruction find(Long id) throws BusinessRuleNotSatisfied;
}
```

We map a PaymentInstruction domain object to an object that is suitable to be stored in the database using the dao.PaymentInstructionFactory This factory relies on Dozer to map it into an entity which is enriched with ORM annotations:

```
public PaymentInstructionEntity createEntity(PaymentInstruction pi,
    OrderingAccount acct) {

        PaymentInstructionEntity pie = DozerBeanMapperSingletonWrapper
        .getInstance()
        .map(pi, PaymentInstructionEntity.class);

        pie.setOrderingAccount(acct);

        return pie;
}
```

Editing a payment involves retrieving a Payment Instruction from the database and map it to a domain object again using the same patterns found in the previous defined factory. Finally, the domain object is mapped to a DTO and serialized to the client in JSON format.

6 To conclude

As we demonstrated in this article, DDD can be fairly easy applied in a CRUD application with today's business requirements. We have answered the question why we would use a domain-driven design in our CRUD application. Using DDD we achieve more traceability of requirements because of the isolation of the domain. It improves understanding of the domain by everybody (including developers) and the ubiquitous language aids in templating requirements and the adoption of Behavior Driven Design.

The pattern language helps one to formulate the domain in a structured way and focuses on what is relevant and reaches the right level of abstraction (i.e.: Entities, Value Objects, Domain Services, Modules and Aggregates). Using package private create() methods in our Domain Objects models the dependence between the Aggregate root and the objects in its boundary.

DDD prevents applying anti-patterns like Smart-UI and Fat Service Layer which violate the high cohesion principle. Separating the interface from its technical implementation, the domain layer becomes reusable because it doesn't depend on technical dependencies. This can be further enforced by e.g. the Maven enforcer plugin.

The trade-offs made in applying such a design are the additional front-end machinery that is needed to check business rules each time a dependency changed but it benefits of having the domain logic in one place and not duplicated across front- and back end. Also, implementing equals and hashCode in a domain object violates the isolation principle somewhat but it is not trivial to extract these two methods from the object on which these methods need to be implemented.

References

[1] Eric Evans. *Domain-Driven Design, Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[2] https://github.com/ingmars1709/ddd.git

[3] http://dannorth.net/introducing-bdd/

[4] http://jbehave.org/

[5] KnockoutJS.com

[6] https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

[7] http://www.martinfowler.com/eaaCatalog/separatedInterface.html

[8] https://maven.apache.org/enforcer/

[9] https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

[10] http://dozer.sourceforge.net/