|  |  |
| --- | --- |
| Title: |  |
| Attendees |  |
| Absent |  |

Source: http://sanderhoogendoorn.com/blog/index.php/category/uml/

1. [Modeling your domain models in UML](http://sanderhoogendoorn.com/blog/index.php/modeling-your-domain-models-in-uml/)

Posted on [**May 12, 2011**](http://sanderhoogendoorn.com/blog/index.php/modeling-your-domain-models-in-uml/) by [**shoogend**](http://sanderhoogendoorn.com/blog/index.php/author/shoogend/)

[Reply](http://sanderhoogendoorn.com/blog/index.php/modeling-your-domain-models-in-uml/#respond)

*In any given system under development, the business domain of that system is key. It holds all concepts important to the domain, and captures the business logic from the domain. An important question to answer is what to model about the objects in that domain and how to model this in UML class diagrams.*

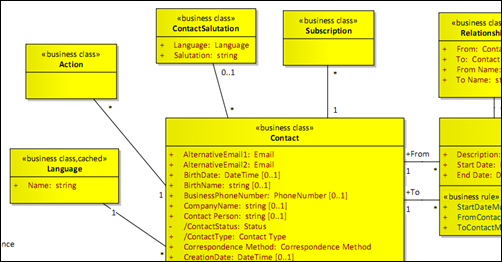
**Identifying the elements of the business domain**

First of all, it is important to underline which types of classes participate in the model, simply referred to as the domain model. Aligned with the paradigm of domain driven design, I recommend to model the following types of classes:

* Domain objects.
* Factories.
* Repositories.
* Services.

**Domain objects**

A domain object captures a concept from the real world of the application. Think of **Contact**, **Contract**, **Company** or **Address**. Domain object are considered to have identity. Next, domain objects have properties, and these properties have types. Also, domain object deliver services (often implemented as methods).

[](http://sanderhoogendoorn.com/blog/wp-content/image92.png)  
*A number of domain objects in a class diagram*

A concept in the business domain should be considered as such if it has either state (modeled in properties) or delivers services (modeled as operations). Or of course both.

**Factories**

Factory classes are used to create new instances of domain objects. Although this can often be achieved by just directly creating a domain object from scratch, in most domains creating new object is more complex, especially with the core concepts in the business domain. For instance, when we created a new **Contact**, it need to have a least one **Address** which also needs to be created, and it should be aligned with the current **Company**. When creating such a construct (often referred to as a aggregate in domain driven design) a factory class is used (and modeled) to holds this creation functionality.

A factory class can cover multiple creation scenarios. Name your factory classes after the domain object it creates. Think of classes such as **ContactFactory** and **CompanyFactory**. Model each of these scenarios as a method on the factory class.

**Repositories**

Additionally to the factory classes, a business domain can have repositories. A repository is responsible for answering enquiries about existing instances of domain objects. For instance, you might want to grab all instances of **Contact** who live in Amsterdam. Name such repositories after the corresponding domain object. Think of **ContactRepository** or **AddressRepository**. Model such enquiries as methods on the repository associated with the domain object. Please note that many projects choose to combine the factory and repository classes.

**Services**

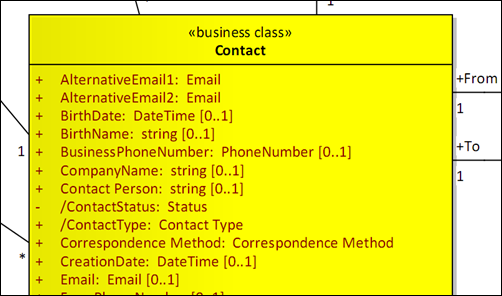
Some services in the business domain do not directly correspond to a domain object, but rather apply to several different domain objects. In this case, such as service can be modeled as a separate class, usually with the name of the actual task as its class name. Think of a service such as **ImportFromExcel** that imports a project, its work items and their estimates into a dashboard.

As an alternative, projects may choose not to model separate factories, repositories and services, but rather add all their functionality as services on the actual domain object. In this case, I would recommend stereotyping these methods for example as **factory method**, **repository method** or **service**, so their purpose is immediately clear.

Please note that even though now these methods are modeled on the domain object, they will likely not be implemented there, but on the actual factories and repositories. This alternative is merely a technique to avoid cluttering the domain model with unnecessary classes. Domain models are full enough without adding factories and repositories.

**Identifying properties**

Now let’s delve down to the domain objects, clearly the most important type of class in the domain model. Next to delivering services, domain objects have properties, such as Name, **Email**, **City**, **Country**, **Level**.

[](http://sanderhoogendoorn.com/blog/wp-content/image93.png)  
*A domain object* ***Contact*** *with a number of public properties*

You will need to identify all meaningful properties on your domain objects. Basically you will need to identify all properties that a domain object exposes to the outside world, either other classes in the domain, user interface elements that bind to it, use cases or persistency layers. These are the public properties of the domain object.

Do not model fields that are internal to a domain object as these are not relevant to the business domain but in most cases are only relevant to the developers. Thinks for instance of the **Id** of a domain object, that is required to store it in a database or to a web service.

A special reminder goes out to properties that are not persisted but are calculated or assembled by the other properties and fields of the domain object. These are called derived properties, and should be marked as such in the domain model.

**Modeling property types**

And each of these properties has a type. These types actually hold quite a lot of information on the business domain – maybe even more than you’d expect.

I consider the following types of property types:

* Basic types such as **string**, **DateTime**, **float**.
* Values types or objects such as **ISBN**, **Email** or **BSN**.
* Enumerations
* Variable lists.
* Associations.

**Basic types**

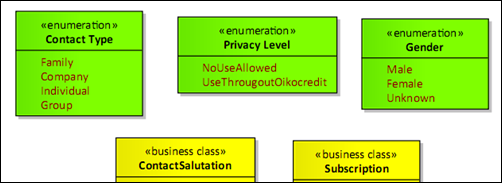
Many properties will appear to have a basic type, such as **number**, **string**, **date**, **timestamp**, **float**. However, note that for a whole lot of properties, you actually quite a lot more than that it’s just a **string**. Therefore, for each property check properly if you know more about it than that it just has a basic type.

**Value types or object**

A value type or object is an object that is recognizable by its value, but where we are not interested in it’s identity. For instance, two instances of my email address will appear to be the same, while two instances of **Company** that can have exactly the same values for all properties can still be two different companies.

**Enumerations**

Sometimes, a property can only have a limited set of values. In this case you might consider modeling an enumeration (which is a standard stereotype in UML). This enumeration holds the possible values, so next you can model the property of the type of the enumeration.

[](http://sanderhoogendoorn.com/blog/wp-content/image94.png)  
*Enumerations* ***ContactType****,* ***PrivacyLevel*** *and* ***Gender****.*

For instance, a domain object **Contact** might have a **Type** property that be **Individual**, **Group** or **Family**. Now model an enumeration **ContactType** that holds these values, and model the property **Type** of type **ContactType**.

But, there’s a restriction to using enumerations. Only use these if you are certain that the list of possible values is static, that is does not change over the course of the life of you application. If you need to add to or change the list of values, this results in re-testing your whole application as the values are often used to perform conditional operations. So if you think the list of values is going to change don’t use an enumeration. Model a variable list instead.

**Variable lists**

If a property can only have a value from limited set of values, but that list is subject of change during the life of your application, enumeration are not the best solution. The list of values itself should be maintained – either by some software, possibly external to your application, or directly in the database. Think of lists of countries, states or academic titles.

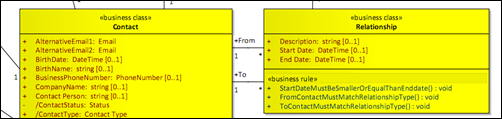
Therefore you will need to take extra care of these properties. Now there’s a number of options here. If the list only uses standard properties (think of **Name** and **Description**) and is just used for reference, you might want to group all these different list into one single domain object (and hence table). In this case, an additional properties is required for this domain object, for instance called **Type**, that describes the type of element, such as **Country**, **State**, or **AcademicTitle**. In this option you will only need a single maintenance use case to maintain all values in all lists. The value or now available for reference, selection or printing on envelopes. We usually refer to this pattern as *Smart Reference*.

For modeling a property of the type of such a variable list there are a number of options. Either refer to the type of list in the name of the property (call it **Country**, **State** or **AcademicTitle**) and model **SmartReference** as the type of the property. This works out fine as long as you don’t have to properties of a domain object referring to the same variable list.

A more agreeable option is to give the property the name you want it to have, and model the type of the variable list as the type of the property. To make sure the property is recognized as a variable list, add a stereotype (for instance **smartreference**) tot the property.

**Associations**

And then there’s the option that would want to refer another domain object. In this case you normally model an association rather than model a property. An association models a structural relationship between two (or more, but don’t go that road) objects in your business domain. So if a contact has zero, one or multiple relationship, you can model this as a one-to-many association between the domain object **Contact** and **Relationship** in your domain model.

[](http://sanderhoogendoorn.com/blog/wp-content/image95.png)  
*Two (named) associations between* ***Contact*** *and* ***Relationship****.*

In code all associations (as in UML) are represented by properties that exist on both classes. So in the example **Contact** will have a property that represents a list of relationships. And vice versa, each **Relationship** will have a property of type **Contact**. Unless you add additional names to the ends of the association in the domain model, the names for these properties are undefined. However it’s a good habit to name them after the domain object they come from. So the properties will simply be called **Contact.Relationships** and **Relationship.Contact** – mind the plural.

However, if there’s two different associations between the same domain objects, applying this default naming patterns isn’t enough. It would result in multiple properties with the same name. In that case always specify meaningful names explicitly to the ends of the association. For instance if another association (one-to-one) between **Contact** and **Relationship** exists that identifies who’s the from and who’s to to in the relationship, this could result in properties **Contact.Froms** (a list of type **Relationship**) and something like **Contact.Tos** (also a list of type **Relatioship**).

Another reason for specifying meaningful names to association ends is that these simply occur in the vocabulary of the business domain.

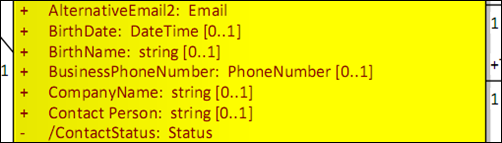
**Adding validations**

The next step is to add validations to you domain model, and your domain objects in particular. I’ll look at the following types of validations:

* Are properties *mandatory* or not?
* Is the property valid if the type is a *value object*?
* Is the property correctly *formatted*?
* Does the domain object adhere to the defined *business rules*?

**Mandatory properties**

Most properties on domain objects will be mandatory (although debatable). Therefore by default the multiplicity for a property is set to **[1..1]**, which means that there is to be exactly one occurrence for this properties on each object. If a property is not mandatory for the domain object to be valid the multiplicity of such a property is set to **[0..1]** – which implies that zero or one instances of this property are present.

[](http://sanderhoogendoorn.com/blog/wp-content/image96.png)  
*Mandatory and non-mandatory properties*

In the example above, **BirthName** is not mandatory, but **AlternativeEmail2** is.

**Value objects**

The use of value types of objects in your domain objects explicitly adds validation to your objects. It means that the validations rules for this particular value object will apply to such a property automatically. If a property is defined using type **Email**, all values (when checked) need to follow the rules defined on the value object Email that define when something is in fact a valid **Email**. Of course, these validations can best be modeled on the value object itself.

[](http://sanderhoogendoorn.com/blog/wp-content/image97.png)  
*Properties that have a value object as their type (****Email****)*

In code, this could results in the following property for class **Contact** (using the ADF.NET framework).

1: public Email AlternativeEmail2

2: {

3: get { return state.GetValue<Email>(ContactDescriber.AlternativeEmail2); }

4: set { state.SetValue(ContactDescriber.AlternativeEmail2, value); }

5: }

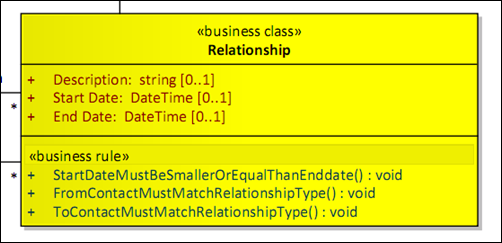
**Formatting**

Some properties follow a certain format. Think of properties representing money or dates. If the type of the property is a value object, the format should be specified there (often as a regular expression). If not, model the format with the property. Please note that formatting is often culture specific.

Formats can include aspects such as length attributes, for instance a string may not be longer than 64 characters, or must be within a certain range, for instance when a value must be larger than 0 but smaller than 128.

**Business rules**

Additionally domain objects can validate business rules. For instance on a domain object **Relationship**, a business rule may validate that both contacts in the relationship match some relationship type. This can be defined in an operation (or a method in code) called **ToContactMustMatchRelationshipType**. Although this seems like an abundant long name, in fact I recommend to name business rules exactly to what they’re actually doing. Moreover, I recommend to implement the business rules in code using exactly the same name, thus creating traceability.

[](http://sanderhoogendoorn.com/blog/wp-content/image98.png)  
*Modeling business rules in your domain object using the* ***business rule*** *stereotype*

I also recommend to add the stereotype **business rule** to these operations to distinguish them from other methods your classes have. In code the business rule **ToContactMustMatchRelationshipType** could look something like this. Here the attribute **BusinessRule** is attached to the method for the same reason, and to allow for automatic validation.

1: [BusinessRule]

2: public ValidationResult ToContactMustMatchRelationshipType()

3: {

4: return !To.IsEmpty && RelationshipType.To != To.ContactType

5: ? ValidationResult.CreateError(this.Prop(r => r.To), "Contact type '{1}' for {0} does not match required contact type '{2}'.", To, To.ContactType, RelationshipType.To)

6: : ValidationResult.Success;

7: }

An interesting type of business rule is that where the domain object at hand is validated against other domain object (mostly of the same type). For instance, when adding a new **OrderType** object, even though it is by definition unique (as domain object have identity by nature, usually implemented using an **Id** property), may not have the same name as any of the other already existing order types. Or a new contract

**When to validate?**

An intriguing question that comes to mind when discussing validation is when to validate the validations? In a traditional Windows user interface one might say that the moment the user leaves a field and moves to the next the validation for the represented domain object property could or should be fired. In a web application typically the user types in all the fields and then presses submit.

Validating the objects at hand should fire all validations, value objects, mandatory properties, formatting, business rules that are valid for the event (such as save, remove). In the code example from use case **Manage Address** an **Address** object is validated and than saved .

1: public void SaveAddress()

2: {

3:

4: if (this.Execute(

5: () => Address.Validate(),

6: () => Address.Save(),

7: () => SaveContact()))

8: {

9: OK();

10: }

11: }

Posted in [.Net](http://sanderhoogendoorn.com/blog/index.php/category/net/), [C#](http://sanderhoogendoorn.com/blog/index.php/category/c/), [Design patterns](http://sanderhoogendoorn.com/blog/index.php/category/design-patterns/), [Domain driven design](http://sanderhoogendoorn.com/blog/index.php/category/domain-driven-design/), [UML](http://sanderhoogendoorn.com/blog/index.php/category/uml/) | [Leave a reply](http://sanderhoogendoorn.com/blog/index.php/modeling-your-domain-models-in-uml/#respond)