Getting Started With UML Class Modeling

An Oracle White Paper
May 2007

ORACLE

INTRODUCTION

UML class modeling is one of the major UML modeling techniques. It is typically used to detail use cases (see also the white paper "Getting Started With Use Case Modeling") and to provide a first-cut of the design of a system.

This paper discusses the modeling elements that can be used in a UML class model and describes how class models can be developed iteratively using JDeveloper 10.1.3, and Oracle Consulting's best practices with that. For those familiar with Structured Analysis a comparison is made with Entity Relationship Modeling.

WHAT IS CLASS MODELING

The words "class" and "object" are often used as if they are the same, but actually they are not. An **object** is someone or something, like the person "John" or this document. It can be concrete like "The Eiffel Tower" or abstract like "France". Formally put, an object is something with properties, relationships, behavior and an identity.

A class is a definition of objects that share the same properties, relationships and behavior. An object is an instance of a class. The properties of a class are called attributes and the behavior is expressed in operations.

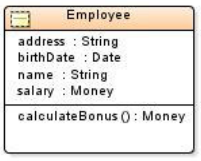
A **class** is a definition of objects that share the same kind of properties, relationships and behavior, like the class Employee may define the properties "name", "age" and "employee number" and may define a relationship with the class Department. An object is a specific **instance** of a class, like "John" may be an instance of the class Employee.

In the context of system development we define classes with properties and behavior that are relevant for the System under Discussion (SuD). Class Modeling is the task of specifying these classes using a specific language in which the properties are called **attributes**, the relations are called **associations** and behavior is defined as **operations**.

UML class modeling is platform independent, so it is not about Java, nor C#. At some point classes are transformed to a platform specific technology, which may be Java classes, Enterprise Java Beans, ADF Business Components and so on. However, as will be discussed later, there are there situations in which it makes sense to let the UML class model anticipate constraints of the technology that is going to be used.

CLASSES, ATTRIBUTES AND OPERATIONS

A class is drawn as a rounded rectangle like in the figure on the right.



The Employee class

The rectangle can be divided into three compartments, with the name in the upper, the attributes in the middle and *operations* in the lower compartment.

There are several UML concepts that are drawn using the rectangle shape. The exact concept is displayed using a guillemet quoted name (<< >>). By default JDeveloper shows the class concept in the diagram as "`<<class>>`", as well as the package, for example as "`mypackage::model`".

However, as the class is the most common concept, displaying this keyword is often suppressed. Therefore, consider configuring JDeveloper to hide the concept name and the package of classes, to prevent them from cluttering the diagram. Hiding the concept name and package can be done in the menu by going to Tools -> Preferences, expand the nodes Diagrams -> Class. You can edit preferences at different levels, from the whole diagram to a specific concept. When you want to change this for classes only, you select Class from the drop-down list and uncheck Show Package and Show Stereotype from the Display tab. You can also set this for individual shapes by right-clicking them and select Visual Properties from the context menu. If you want you can do the same to hide the operation and attributes sections for classes that do not have operations or attributes.

The names of classes start with an uppercase. The names of attributes, operations and arguments of operations start with a lower case. There are no spaces or underscores between words, while each new word starts with an uppercase.

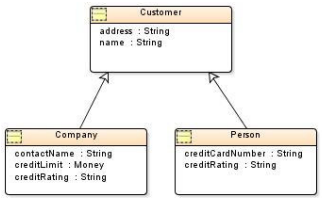
It is a de facto standard to write class names with no spaces or underscores between words, each word starting with an uppercase, as in "OrderLine". Attribute names and operations start with lower case, while each new word starts with uppercase, as in "contactName".

An attribute has a type, which is put after the name, as in "contactName : String". Operations can take arguments and may return an object of a specific type. In "doSomething(int someArgument) : String", an operation with name "doSomething" takes an argument with name "someArgument" of the type "int" and returns an object of the type "String". When no object is returned, this is indicated using the "void" keyword, as in "doSomething() : void". As you can see, the operation in the latter example takes no argument either.

GENERALIZATION

A generalization is a relationship between a more generic class, for example *Animal*, and a more specific class, for example *Dog*. The generic class *generalizes* the specific one as well as the specific one *specializes* the generic one. A generalization can be interpreted as an "is-a" relationship. The generic class is called the **superclass** whereas the specific class is called the **subclass**.

In UML a generalization is indicated by drawing an open arrow from the subclass to the superclass. The subclass is preferably put below the superclass to express the hierarchy as in the example to the right where Company and Person are subclasses of the superclass Customer.



Generalization

A class can generalize several subclasses that are then said to specialize the superclass. Subclasses inherit the attributes and behavior of the superclass but may override that. Proper usage of inheritance can be validated by applying the Principle of Substitutability.

A subclass inherits all attributes, relationships and operations from the superclass. Generalization/specialization is therefore also referred to as **inheritance**. The subclass can have specific attributes, relationships and operations and may override the ones of the superclass.

In UML a subclass can inherit from more than one class, which is called **multiple-inheritance**. Multiple-inheritance often is not supported at implementation level. Java, for instance, does not support multiple-inheritance for classes. The rationale is a potential risk of conflicts with multiple-inheritance. Suppose that class C inherits from both classes A and B that both have an attribute "address" or an operation "doSomething()" which are not equally defined. What does that mean for class C, which address should it inherit or which one of the two doSomething() operations should it execute? To prevent the need to solve this type of problem during implementation, the class modeler of JDeveloper does not support multiple-inheritance. Whenever this becomes an issue can it be solved by using an interface instead (to be discussed later).

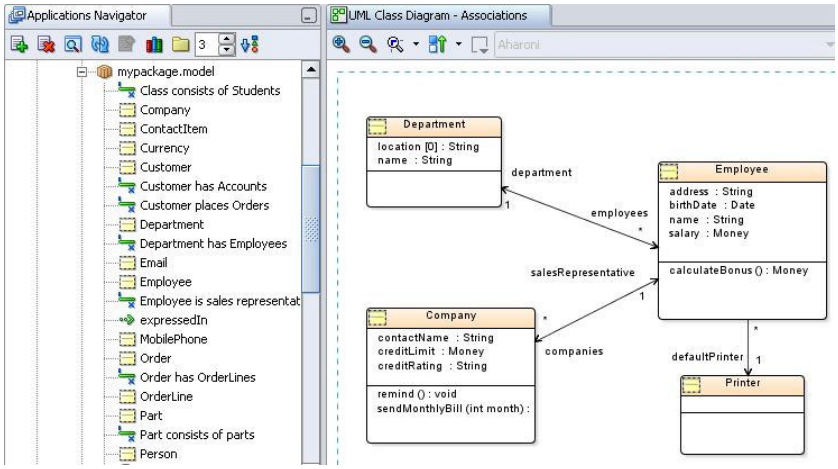
Use generalization where relevant for the System under Discussion but avoid overgeneralization. For example, that an Employee and a Customer both have a name and an address does not automatically imply you should make them subclasses of a Person superclass. You should only do this when both an Employee and a Customer 'are-a' Person, and the Person class means something from a conceptual point of view.

To recognize improper usage of generalization, you can apply the **Principle of Substitutability**, which states that objects (instances) of the subclass may be used everywhere objects of the superclass appear, but not vice versa. In this case this means that everywhere Person objects appear you should be able to replace them by Employee or Customer objects.

ASSOCIATIONS

An association is a structural relationship between classes that specifies that objects of one class are connected to objects of another class, like Employees are connected to a Department.

Associations are drawn in UML as solid lines. At both ends of the line the **multiplicity** of the association is indicated, which expresses how many objects of one class are associated with how many objects of the other class. In the following example an Employee is associated with exactly 1 Department, whereas the Department can be associated with zero or more (*) Employees.



Classes and associations in the navigator

The symbols used for multiplicity are:

0..1	Optional (zero or one)
1	Required (one and only one)
*	Zero or more
1..*	One or more

As an alternative to *, 0..* can be used.

JDeveloper requires an association to have a unique name, and will assign a default. It is good practice to start the name of the association with the name of one of the classes and let it express the nature and multiplicity of the association. In the example above has the association between Department and Employee been named as "Department has Employees". As you can see the association is located in the navigator directly below the Department class. Which class you start with is arbitrary as long as you do it consistently, for example always from the one-side to the many-side.

Associations describe the relationship between classes. At both ends of an association you indicate the multiplicity and the role name.

Each end of an association has also a name that represents the **role** of that association end. You use the same naming conventions as is used for attributes and you let it express the multiplicity, like in the example the association between Employee and Company has been named "salesRepresentative" at the Employee side and "companies" (plural) at the Company side.

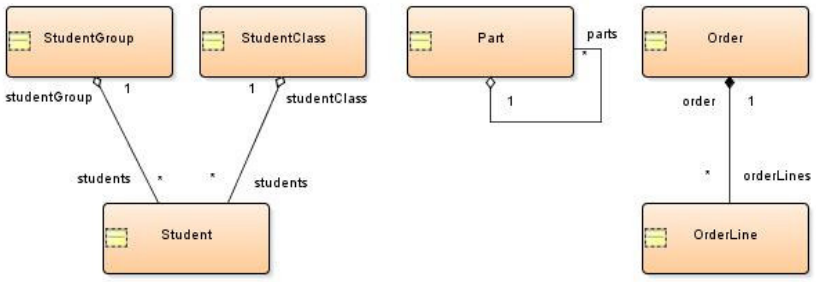
In diagrams you normally show the role names instead of the name of the association, as role names add more value to understanding the diagram. Consider configuring JDeveloper likewise. You can do that in a similar way as has been described in "Classes, Attributes and Operations". At the end you select the Association node instead of the Class node and you uncheck Show Association Name and check Show Role Names.

In UML you can also indicate the **navigability** of associations by drawing it as an arrow. In the example above one of the associations has been drawn as an arrow from Employee to Printer and not vice versa. This indicates that you are able to "ask" an Employee what its default printer is, but you cannot "ask" a Printer for which Employees it is the default printer. An association that is only navigable to one side is called **unidirectional** whereas an association that is navigable to both sides is called **bidirectional**.

Aggregation and Composition

Aggregation and composition are both a "whole-part" relationship. In case of a composition, the part cannot exist without the whole whereas in case of an aggregation it can. Aggregation therefore is often called "weak aggregation" while composition is called "strong aggregation".

In UML an aggregation is drawn as an association with an open diamond at the side of the "whole", as has been done with the association between Student Group and Student in the example below. A composition is drawn with a closed diamond at the side of the "whole", as has been done with the association between Order and Order Line.



Aggregation versus composition

More on Aggregation

Aggregation implies there can be no circular relationship. In the example above, the reflexive association of a Part indicates that part A can be a part of some other part B but not of itself. A class can participate in more than one aggregation, like Student in the example participates in two aggregations: one with Student Group and one with Student Class.

More on Composition

The "whole" in a composition determines the lifespan of the "part". In general a deletion of the "whole" is considered to imply a cascade delete of the "parts", unless specified otherwise (for example as a *constraint*, which will be discussed later).

Composition does not imply that a "part" cannot be transferred from one "whole" to the other, but it is the responsibility of one "whole" to provide the "part" to another "whole". A class can participate in only one composition.

Composition or Aggregation?

To understand the difference between composition and aggregation, review the examples of the previous figure. A Student Group consists of Students. A Student does not cease to exist when the Student Group is dismantled. You therefore model this as an aggregation. An Order consists of Order Lines. When the Order is deleted, the Order Lines will vanish with it. You therefore model this as a composition.

Aggregation and composition both represent a whole-part relationship In case of composition does the lifespan of the whole determine the lifespan of the parts. Only use aggregation when its meaning is clearly understood.

Whether you should use aggregation or composition may depend on the context of the System under Discussion. Suppose you want to model that a Computer consists of parts, among them a Hard Drive. For an organization that sells computers and supplies you probably model this as an aggregation as a hard drive can be sold separately. But for an organization that only *uses* computers, you might decide to model this as a composition because the fact that a hard disk can be removed from a computer is totally irrelevant for the organization.

Martin Fowler (author of "UML Distilled: A Brief Guide to the Standard Object Modeling Language" see also "More Information" at the end) suggests to use aggregation only if you feel yourself a fancy UML modeler. James Rumbaugh (one of the original authors of UML, together with Ivar Jacobson and Grady Booch) even calls it a "modeling placebo". In general the concept of composition does add value, but you can leave aggregation out of a model (that is: model it as a normal association) without things going wrong. Mind that in that case you should document non-circular reflexive associations as a constraint.

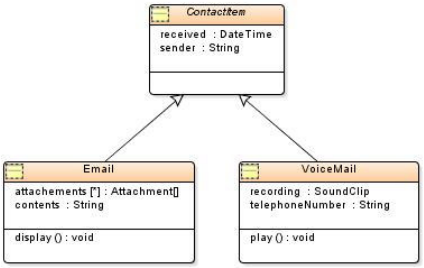
OTHER ARTEFACTS AND CONCEPTS

Abstract Classes

An abstract class cannot be instantiated. The opposite is a concrete class.

An abstract class is a superclass that cannot be instantiated. Put in other words: there are no objects for that class, only for the subclasses. An abstract class can have attributes, associations and operations like other superclasses. When a class is not abstract, it is said to be **concrete**.

An example is the class Contact Item that is a superclass for the concrete subclasses Email and Voice Mail, assuming that for the System Under Discussion there is conceptually no such thing as a "contact item". In UML an abstract class is indicated by putting its name in italics, as in the figure on the right.



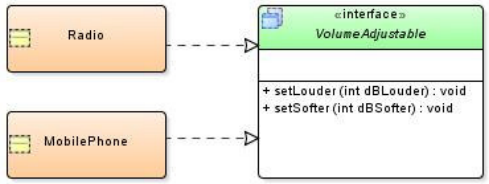
Abstract class

Interface

An interface is somewhat similar to a superclass. The difference is that a superclass represents an "is-a" relationship whereas an interface represents a "behaves-like" relationship.

An **interface** is a collection of related attributes and operation signatures. A class can implement or **realize** these attributes and operations. Whereas inheritance from an (abstract) class models an "is-a" relationship, does the realization of an interface model the "behaves-like" relationship. Consider an interface as some kind of a contract to implement specific standard behavior, while hiding the specific implementation of this behavior. Interface names often end with "-able" or "ible".

For example a Radio and a MobilePhone may both implement the interface VolumeAdjustable, meaning they both can behave as something of which you can adjust the volume. But generally a Radio and a MobilePhone are not considered to be both some "kind of" generic class of which you can adjust the volume, making modelling it as a generalization inappropriate.



The VolumeAdjustable Interface

In UML an interface is indicated with an italic name and a dashed arrow from the implementing class to the interface. Furthermore is it customary to put the interface to the right of the implementing class, as in the example on left.

To distinguish interfaces from classes, the display properties of an interface can be set to have a different color and to display the concept name `<<interface>>`, as has been done in the example above.

An interface can be subclassed but only by another interface. It can also have an association, but only with another interface.

As indicated before, an interface can be used to compensate not having the ability to specify multiple-inheritance. For instance, suppose you want to model the class Employee as a subclass of the Person superclass, inheriting attributes like "name" and "address" and you also want to model it as a subclass of the User superclass, inheriting attributes like "userid" and "password". Modeling either Person or User as a superclass and the other one as an interface can solve this. Using the already mentioned Principle of Substitutability can help to decide what to do.

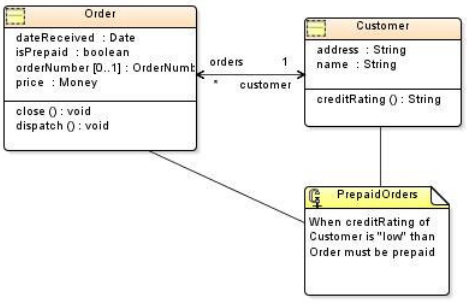
Constraint

In the context of class modeling a constraint can be defined as a restriction that applies to the state an object or a set of objects (not necessarily of the same class) can be in, or the transition of one state to another. An example of a simple constraint is that the name of each Employee must have a value. A more complex constraint is that a Customer with a low credit rating must pay all Orders in advance.

Many constraints are represented by the model as structural assertions. Other constraints can be documented by using UML constraints. The nature of the constraint is preferably expressed using natural language.

Many constraints can be represented in the structure of the class model itself. For instance, the association between Employee and Department represents that for each Employee the Department must be indicated. Another example is that you can use aggregation to express that a Part cannot be a part of itself, as has been done in the example in the section "Aggregation and Composition".

Constraints expressed in the structure of the class model are often called **structural assertions**. Other constraints can be documented by using UML **constraints** that are drawn like the yellow symbol in the example on the right. Constraints can be connected to the classes they relate to, using the "Attachment" component from the palette of the UML class modeler.



The PrepaidOrders constraint

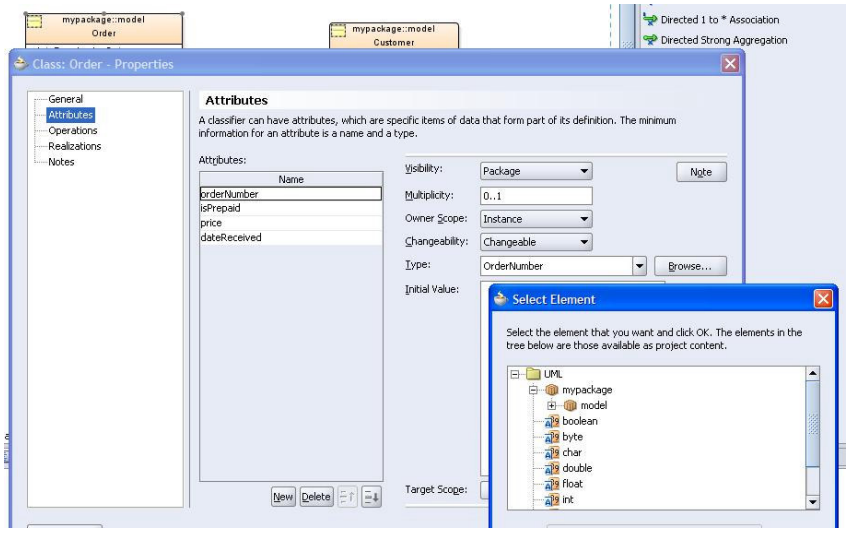
Constraints can be expressed using natural language or by using more formal ways like for example the Object Constraint Language (OCL), which is a formal, mathematical based language and a part of UML 2.0. OCL is not simple to understand and currently only few people are familiar with it. Therefore, as long as constraints specified in OCL cannot be transformed automatically into some implementation, you probably are best of using natural language.

ADVANCED PROPERTIES

This section discusses some of the advanced properties of attributes and operations.

Advanced Attribute Properties

The **Type** property of an attribute is by default String. You can change it on the Attributes property tab of the class as shown hereafter. You can select types from the poplist, select an "element" by using the Browse button to the right of the poplist, or key in a type yourself.



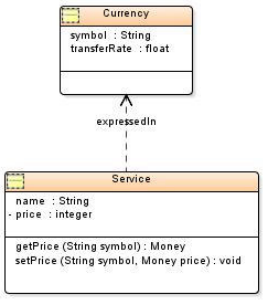
Setting the type using the Select Element button

Mind that a class model is not an implementation model. Therefore, rather than using implementation types as oracle.jbo.domain.Number you use types like String, Integer, Date or Float. You can also use user-defined types, for example Photo or Money, as long as the meaning of the user-defined type is clear to the reader of the diagram.

You can specify the **Multiplicity** property of attributes. The default is 1. You can distinguish optional from required attributes by setting the multiplicity to 0..1. You can also set the multiplicity to [0..*] or [3..5] for example, indicating that the attribute consists of an unlimited array of the indicated type or an array that has at least 3 and a maximum of 5 elements respectively.

You can use the Multiplicity property of an attribute to indicate that it is optional. You can prevent attributes from being accessed by other classes by adjusting the Visibility property. If and when an attribute can be updated, can be specified by the Changeability property. The default of an attribute can be specified using the Initial Value property.

It is a generic object-oriented principle to assume that the data of a class never is accessed directly. This principle is called **data hiding**. For example you get and set the value of the attribute "name" using a getName() and setName operation. You do not need to specify those operations as they are there implicitly, unless the visibility (see hereafter) specifies otherwise. However, in the example to the right the price of a Service is expressed in some currency independent value.



Private attribute

Instead of getting and setting the price attribute directly you are supposed to use the specific operations getPrice() and setPrice() that take the symbol of the currency as an argument and derive the price from that.

For this reason the **Visibility** property of the price attribute has been set to "Private(-)" which has been indicated by the "-" symbol in front of the attribute name. Other values of the Visibility property are "Public(+)", which is the default, "Package" and "Protected(#)". In case of "Package" the attribute is only accessible from other classes in the same package, in case "Protected(#)" only from the class and every class that (indirectly) inherits from it. Mind that because of the data hiding principle you in fact adjust the visibility of the implied getter and setter operations for the attribute.

The default value of an attribute can be specified by setting the **Initial Value** property. If and when the value of an attribute can be updated can be set by the **Changeability** property, which can have the values "Changeable" (updateable, which is the default), "Add Only" (insert only) or "Frozen" (non-updateable).

You might have a need to display attributes in a specific order, for example the most important ones at the top, the least important ones at the bottom and attributes that belong to each other (like startDate and endDate) together. Unfortunately you cannot sort attributes with JDeveloper, they will always be displayed alphabetically.

Advanced Operation Properties

You can enforce that subclasses need to provide a specific implementation of an operation by marking it as abstract at the superclass level. Operations can be prevented from being called by other classes, by adjusting its visibility.

The **Abstract** property of an operation of a superclass is checked to indicate that there is no logical implementation at the level of the superclass but at the level of each individual subclass instead. An example is the superclass Account that has an abstract operation getTotal(), which returns the value of the Account. Account is subclassed by the classes SavingsAccount and StockAccount that each provide a specific implementation of the getTotal() operation.

You can use the **Visibility** property of an operation in a way similar to the usage for attributes.

DEVELOPING IN ITERATIONS

Classes are normally developed in iterations. The following discusses how such an iterative development might look like.

Domain Model

While specifying the requirements you create a **domain model** with **domain classes** that captures the most important classes for the System under Discussion[1]. The domain classes thereby detail the requirements that have been captured with use cases (see also the white paper "Getting Started With Use Case Modeling").

A domain model is expressed in the language of the business and generally is restricted to containing the most important classes with relation associations between them and attributes and operations added, but without using any of the special modeling constructs.

The domain classes represent "things" or "events" that exists in the environment in which the System under Discussion works. The language that is used is the language of the business.

Generally you specify only the most significant attributes. The relationships between classes are specified as associations with multiplicity; usually you do not (yet) use generalization, aggregation or composition. The navigability of an association is also not (yet) specified. You can add operations to domain classes. Interfaces are very rare in a domain model, as is the usage of advanced attribute or operation properties.

Analysis Model

After finishing the domain model, the requirements are analyzed to get a better understanding of these requirements and to describe how the system should support them. This results in the **analysis model** with **analysis classes**. The requirements are structured while focusing on maintenance, like resilience to change and reusability.

An analysis model is expressed in the language of the developer and focuses on maintenance. However it still is a conceptual model, meaning it should abstract from implementation details, especially those that address the non-functional requirements.

The language used to describe the analysis model is the language of the developers. But, being a conceptual model, the analysis model is supposed to abstract from specific technologies or languages. However, whenever suited for the situation you can adjust the analysis model to take constraints of the technology that is going to be used into consideration. For instance, suppose JDeveloper would have supported multiple-inheritance but the technology to be used is Java (which does not support multiple-inheritance). And suppose that JDeveloper would lack the option to indicate which UML superclass to transform to a Java interface and which one to a Java superclass. It would then be sensible to explicitly model one as a UML superclass and the other as a UML interface.

During this analysis you typically start applying **design patterns**. Design patterns capture solutions that have been developed and evolved over time. Applying them makes your model more flexible, modular, reusable and understandable. Design patterns were popularized by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, who since then came to be known as the "Gang Of Four". The design patterns they describe are generally referred to as the GoF Patterns.

---

[1] The Unified Process distinguishes between *business modeling* and *domain modeling*, resulting in a *business model* or *domain model* respectively. To keep it simple this paper only uses *domain model* to mean both.

Another important set of design patterns, especially for the analysis model, is the GRASP Patterns (General Responsibility Assignment Software Patterns) as has been documented by Graig Larman (see also "More Information" at the end). The patterns and their usage are outside the scope of this paper.

Analysis Classes Stereotypes

Regarding analysis classes a distinction can be made between the entity class, boundary class and control class.

Entity classes are responsible for providing the business functionality. They carry the data that generally is persisted in, most likely but not limited to, a relational database. Examples are the classes Employee, Customer and Order. Entity classes typically map to (EJB) entity beans, (ADF) business components, POJO's that are mapped with TopLink and ("at the bottom") database tables.

Boundary classes are responsible for handling the interaction with the users or external components. Boundary classes typically map to JSP or JSF pages or for example web services.

Entity classes are responsible for providing the business functionality. Boundary classes are responsible for the interaction with users and external components. Control classes are responsible for controlling the flow of events.

Control classes are responsible for controlling the flow of events. They transfer the control to the appropriate entity classes depending on the input received from the boundary classes. A typical kind of control class is the one that coordinates the scenario of a particular use case, with an operation for every happy and stand-alone alternative path. The control class controls the activity and is responsible for handling the transaction. Examples of control classes are Order Handler, or Trade Validator in stock exchange system. Control classes typically map to (EJB) session beans, (ADF) application modules or POJO's.

The distinction between entity, boundary and control classes corresponds with the J2EE **Model-View-Controller** (MVC) design pattern[2].

You cannot stereotype classes in JDeveloper as an entity, boundary or control class. Therefore consider distinguishing them by using different colors.

Creating Analysis Classes

You start the set of analysis classes as a copy of the domain classes from the domain model. This can best be done by copying the JDeveloper project that contains the domain model to a new project.

After that you start adding classes that have not been recognized in the domain model, but are needed for the system to support the requirements. Missing attributes are added and reference classes are created from attributes, like Country or Order Status. You add generalizations or interfaces to the analysis model where appropriate.

---

[2] Strictly speaking MVC is not a design pattern, but an aggregation of several design patterns, like the Façade, and Command patterns and so on.

Carefully consider whether an attribute is really a property of the class or that it should belong to some other class. Review the names of the classes, interfaces, attributes and association roles. Choosing the right name is a good aid in determining the correct responsibilities for classes and a very important part of documenting the model. You are likely to discover new attributes and classes while doing so.

In the analysis model you can use any special modeling construct that is appropriate, like generalization, aggregation, composition, interfaces, and advanced attribute or operation properties.

Having identified all classes with their attributes and associations, you start reviewing the associations and use composition or aggregation where appropriate. Consider making a superclass abstract. You set the type and multiplicity of the attributes and add operations where applicable. Give operations names that clearly express their purpose.

Furthermore, the visibility of attributes or operations can be specified (or any other advanced attribute or operation properties that have not been discussed in this paper).

At some point you need to consider the coupling between classes to support a proper modularization or packaging of the model. Where applicable you make associations unidirectional to enforce a loose coupling and avoid that packages depend on each other in both ways.

Navigability is specifically important in the design model (to be discussed hereafter). You can let JDeveloper transform the analysis model into a design model consisting of Java classes or ADF Business Components. JDeveloper thereby uses the navigability to determine if it should create an accessor attribute to navigate to the other class. For instance, when transforming the analysis classes Department and Employee to Java classes, the Department class gets an attribute "employees" and the Employee class gets an attribute "department"[3]. Deciding upon the navigability *before* the analysis model is transformed has the advantage that you do not need to add or remove accessor attributes manually during design.

You should consider the navigability of associations of analysis classes. An association that is navigable to one side is called unidirectional as opposed to bidirectional

The short recipe for creating analysis classes from domain classes is as follows:

1. Finalize the attributes and create new classes
2. Apply generalizations
3. Add interfaces
4. Apply composition and aggregation
5. Set the multiplicity of attributes
6. Add operations to classes
7. Set superclasses to abstract where appropriate

[3] As a matter of fact, from the perspective of one class there is conceptually no difference between an attribute that serves as an accessor to another class or any other attribute.

8. Adjust the visibility of attributes and operations
9. Set other advanced attribute properties (like Initial Value).
10. (Re)package the classes and set the navigability of associations

Design Model

The analysis model provides the first cut of the **design model**. The design model consists of **design classes** and incorporates decisions that are made to address the non-functional requirements, like performance, distribution and so on. The design model is specified in the language that is used for the implementation, like Java, Enterprise Java Beans (EJB), ADF Business Components, relational tables and so on. Especially during design you will use many design patterns. Design modeling is out of the scope of this paper.

UML CLASSES VERSUS ENTITY RELATIONSHIP MODEL

A UML class model is the object-oriented equivalent of the Entity Relationship Model in the traditional Structured Analysis, though there are some notable differences.

First of all Entity Relationship Modeling is restricted to data modeling, where in UML this is done with the domain and analysis entity classes. There is no true equivalent for boundary and control classes in Entity Relationship Modeling. On the other hand is there not much need for a similar concept with Structured Analysis, as the transformation from analysis to design is fundamentally different, if not simpler. The comparison hereafter therefore restricts itself to the data modeling area.

A class can be compared to an entity and an association to a relationship between entities. Unlike an entity, a class can have operations that are used to document specific behavior of the class. In case of an entity one is forced to specify this behavior elsewhere, for instance by documenting it in one or more functions. Common examples are calculations.

Compared to Entity Relationship Modeling allows UML class modeling for expressing more structural assertions, resulting in lesser constraints.

The constructs one can use with class modeling are richer than with entity modeling. Entity subtyping is similar to inheritance but with limited capabilities, as there is no such thing as an abstract superclass or the possibility to specify the visibility of attributes. After two levels does entity subtyping loose its effectiveness. There is no equivalence for multiple-inheritance with entity modeling and there are no interfaces.

There is also no such thing as aggregation or composition with entity modeling. As has been discussed, aggregation should be used with care, but compositions are often found in class models. With entity modeling one must compensate this by specifying constraints that express how the life span of the "whole" relates to that of the "part".

Generally speaking class modeling offers more options to specify structural assertions than entity relationship modeling and therefore requires less constraints.

There is one exception though. There is no equivalent of the relational **unique identifier** with class modeling. There used to exist the "candidate key" concept in James Rumbaugh's OMT[4], but apparently it did not make it into UML. The probable reason is that the three amigos (as Rumbaugh, Booch and Jacobson are called) decided that objects have an identity by definition, but that still leaves that a unique identifier is a logical concept that people find natural to talk about. Cars have unique license plates, people have unique social security numbers and so on. The concept of stereotypes in UML as such supports extending it to include unique identifiers, but JDeveloper does not yet support defining new stereotypes.

In case of ADF not having a unique identifier imposes an issue. An ADF entity object must have a primary key. As a result you will run into problems when automatically transforming UML classes to ADF business objects, because the transformer will use the first attribute it encounters as primary key. In most cases it's the wrong one, resulting in the need to fix the business objects model. This is especially a nuisance when the primary key plays a role in a one-to-many association, as the wrong attribute will have been transformed into a foreign key at the child side. You need to remove the association, fix the primary key and recreate the association again. Therefore, currently it is not advised to automatically transform a class model into ADF entity objects.

Compared to Entity Relationship Modeling does UML class modeling also help focusing on the problem domain rather than the solution.

But in spite of this all class modeling has its advantages over entity relationship modeling. The advantages not only consist of the fact that you can express more structural assertions. In my personal experience I also found that with entity relationship modeling I have a tendency to "think in tables". With class modeling I have not, which helps me focusing on the problem domain rather than thinking in solutions. More than once I found myself creating a class model, probably superior to the model I would created otherwise when using entity relationship modeling.

MORE INFORMATION

A good book about UML "UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)" by Martin Fowler and Kendall Scott (ISBN 978-0321193681).

The GoF Patterns are documented in "Design Patterns, Elements of Reusable Object-Oriented Software" by Erich Gamma et al (ISBN 978-0201633610). The GRASP Patterns are documented in "Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design" by Graig Larman (ISBN 978-0137488803)

The official UML site can be found at http://www.uml.org/. A short and pragmatic reference on how to use UML modeling techniques, can be found at the

[4] OMT stands for Object Modeling Technique and is one of the predecessors of UML.

site of Scott W. Ambler:
http://www.agilemodeling.com/essays/umlDiagrams.htm

Other papers in the Getting Started With series are:

Getting Started With Use Case Modeling
Getting Started With Activity Modeling
Getting Started With Unit-Testing
Getting Started With CVS

Getting Started With Use Case Modeling
May 2007
Author: Jan Kettenis

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2005-2007, Oracle. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.