BEYOND CORBA

MODEL DRIVEN DEVELOPMENT

Bjørn Nordmoen
Oslo Technology Center
Schlumberger House,
Solbråveien 23,
1383 Asker, Norway
+47 6678 8127
nordmoen@slb.com
Fall 2001

Abstract

This paper will summarize our experiences with **model driven development** from the OBOE project (EU sponsored project). We developed a **component modeling language** (**CML**) from which we could code-generate towards a CORBA based middleware or an EJB based middleware. The CML was based on CORBA IDL also including **the Java-beans event model** and the Object Database Management Group's **(ODMG) relationship model**.

Finally by using the **stereotype** mechanisms in UML, we could represent our concepts in the UML meta-model. By using the scripting facility (visual basic) in Rational Rose, we could then generate the CML model directly from a UML tool. The CML model would then be run through a code-generation tool to produce either EJB- or CORBA middleware code.

This work also relates to the new OMG initiative called "**Model Driven Architecture**."

Introduction

From Solution domain to Problem domain

The first major programming languages (FORTRAN, C) were functional oriented. The concepts were very mathematical and computer near and most problems attacked were relatively simple and of mathematical nature. The main language concept to help handle bigger problems was the subroutine or function. The subroutine helped in breaking a problem into smaller problems. Most of this decomposition and modeling was done in the context of the solution domain.

During the eighties, as the problems one tried to solve grew in complexity, the area of structured analysis and design appeared[1]. The main message in this technique was to structure and model the problem in the context for the problem domain and not in the context of the solution domain. This was definitely a step in the right direction, but still there was a major problem with the impedance mismatch between the problem domain models and the solution domain models (implementation languages).

Fortunately there was a parallel track starting with the programming language SIMULA[2] back in 1967. This language grew out of work related to simulation problems and the main idea was to use concepts that supported the modeling of the real world also in the programming language. This was the start of object-oriented programming the way we know it today, and it also prepared the ground for reducing the impedance mismatch between the modeling of the problem domain and the solution domain because the same modeling concepts could be used in both domains.

During the nineties many object-oriented modeling techniques appeared. Among the more popular were OMT, Boch method and Objectory. Fortunately for the programming community the main players in the object-oriented modeling field joined together and defined a common modeling language called UML[3] in 1996. In fall 1997 UML -1.0 was adopted by Object Management Group (OMG) as an open modeling standard.

UML gave programmers and designers of software systems a common and rich notation, based on the main concepts from object-oriented programming, from which they could model many of their problems. But over the last 10 years, the complexity of the problems to be solved and also the complexity of the solution techniques themselves have grown dramatically and in many ways have outgrown the expressive power in UML. Modern software systems are often

distributed in nature, they need to be able to interoperate with other systems and also work with legacy-systems, and they need to handle persistent data and to run in the context of the world-wide-web.

From Socket Programming to Distributed Object Communication

One of the most significant achievements in software solution technologies in the last decade has been the development in the area of distributed computing, from socket layer programming to distributed object communication frameworks. OMG with its CORBA[4] framework has been a pioneer in this respect. CORBA comes with an interface description language (IDL) from which the communication proxy and skeleton software are code-generated. CORBA also offers support for event handling, relationship and persistence through a set of services that can be used at run-time.

Another significant trend in distributed computing has been the change of client/server systems from simple 2-tier architecture to 3/n-tier architectures, separating the thin client from the business logic and data-storage on the server. Microsoft DCOM/.NET with transaction support, CORBA Components and Enterprise Java Beans (EJB) are all commercial initiatives that try to integrate distributed object technologies with 3/n-tier server frameworks that are popularly called application servers.

From Object-Orientation to Components

A third trend over the past few years has been the emergence of component-based system development. This has been most successful in user-interface developments such as Microsoft's Visual Basic and SUN's Java Beans rapid development toolkits. The technique of building products by assembling a set of other well-defined parts has been proven and used in many other industries for decades. The problem in software is probably the lack of a stable and well agreed upon component architecture. Today's component infrastructures like DCOM/.NET, CORBA Components and EJB are to immature and complex to create a software component market.

In today's business context, with rapid changes in both business needs and technology solutions plus a demand for shorter time to market, the need for a simpler and more integrated approach is clear.

- The concepts from distributed object- and component-frameworks need to be added to the UML models, so the whole system structure can be reflected at the model level instead of hiding this through the implementation-code and the explicit usage of system services.

- The business itself needs to be modeled more in depth and also beyond a single system (business modeling). These models need to be platform independent.

- To enable component-based development there is a need for well-defined reference architectures. The quality of the reference architecture will strongly influence the quality of the system when it comes to key features as interoperability, distribution, maintainability and evolution. There are probably different reference architectures for different problems.

- The complex and time-consuming effort of communication middleware programming needs to be accelerated with code-generation.

- The rapid change and the different flavor of communication middleware needs to be hidden with code-generation. One could generate a platform specific model (PSM) from the platform independent model (PIM) by including platform specific directives.

Most frameworks that exist today are either language, platform or product specific. However, it is useful to be able to abstract away from platform and language issues while creating the system description. The main focus and effort should be on solving the business problems at hand by creating the systems the users need. This paper presents a framework that allows for this abstraction, letting the user specify the system using models and lexical descriptions. These descriptions are then used as input for tools that create mappings toward different platforms, languages and middleware. The framework contains:

- a reference architecture for distributed systems,

- a meta-model which is a UML[3] meta-model extension,

- a component definition language (CML) based on CORBA IDL,
- scripts that generate CML definitions from UML models and
- a code generator that creates CORBA IDL, ODMG ODL[10] and code skeletons.

Model Driven Development

Modeling context – Problem frames

Model driven development is a methodology where all vital parts and aspects of the system under consideration are described with formal models. A key aspect of this technique is to take a holistic approach to system development, i.e. the use of formal models for both the business analysis and the IT-system analysis, believing that this make the mapping both easier and more precise. Another key feature of model driven development is the support of models at different levels of abstraction, from the high-level business models focusing on goals, roles and responsibilities down to detailed use-case and scenario models for the business execution. A third feature is the traceability and refinement relationship between the models as more and more details are added to the system specification when moving from problem space to solution space.

Finally, model driven development also have a clear distinction between platform independent models (PIM) and platform specific models (PSM)[5]. The PSM can be generated automatic or half-automatic from PIM with the support of some platform specific directives. The PIM will be expressed in UML describing the concepts in the problem-domain that also includes concepts from a reference architecture whick is relevant for this problem domain. This is along the same ideas as expressed by Michael Jackson in his recent book "Problem Frames"[6] in which he suggests to organize and classify the problems into well defined groups and then apply a solution patterns that is relevant for that group of problems.

The reference architecture and the modeling concepts

The scope of the OBOE[7-9] project was to focus on distributed business systems and how to specify and build an open component infrastructure for this kind of system. The strategy was to use UML as the modeling language and add concepts needed to specify distributed business systems by using the stereotype extension mechanism in UML. This would then be described as a UML distributed business system profile.

Over the past few years, it has been more and more common to implement distributed systems using a 3-tier architecture, where the system is split into a thin client part and a fat server part. The server part is again split into a business logic part and a data part. This architecture supports the separation of concern when it comes to evolving different part of the system independently.

To build a successful distributed system there needs to be special attention paid to communication performance issues. This leads directly to the granularly problems in distributed systems. The communication protocol between the client and server shouldn't been too fine-grained. Trying to build a system with a uniform distributed object space where all objects could possible be distributed would be a disaster when it comes to performance.

To handle the granularity problems, a service-oriented architecture was chosen. A service is a coarse-grained distributed object that can be looked up in the network through the naming-service. These are the only objects that are network visible. A service object implements a business service or function and export the results to the clients as coarse-grained entity objects. The entity objects are implemented as value objects and are not distributed objects. This architecture allows the design of an efficient client-server communication protocol for the problem at hand. Figure 2 depicts the service-oriented architecture.

Another major problem in building distributed systems is the rapid change of middleware technology and the many available implementation solutions. There is a need to protect the application software from these rapid changes and also make it easier to select the right solutions. This can be done by including the key concepts in an abstract manner into the platform independent model and thereby delay the decision on the actual implementation to a later time.

To support the independence of the communication middleware from the client software, the user-service layer was introduced. This layer is a facade that defines the use-case contract between the client user-interface and the rest of the system. The user-service façade also promotes efficient development by allowing parallel work. The user-interface development can progress independently of the server-side development by implementing a dummy/emulator version of the user-service.

In order to isolate the details of the data-storage subsystem (RDBMS/OODBMS/files) from the service component, a data-layer was introduced. This layer is a façade that hides the database/storage-system APIs from the business logic in the service component. This layer contains persistent entity-objects that reflect abstractions found in the business information model.

The implementation of the persistent entity layer can be done in several ways:

- Using direct binding to the actual data storage;
- Using a de facto database protocol such as ODBC, JDBC or Java Blend or;
- Using relevant services offered by component technologies, e.g. Persistent State Service offered by CORBA.

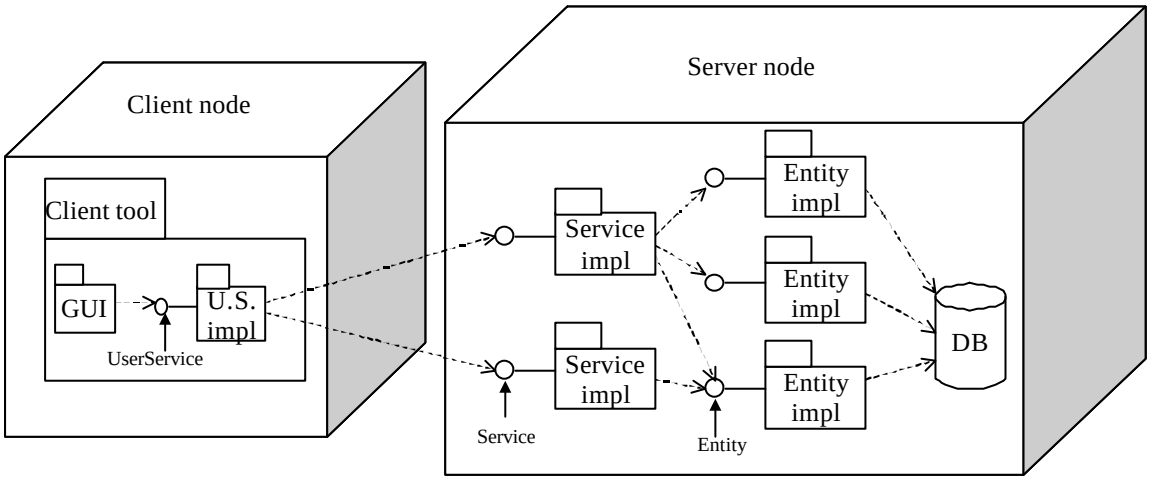

Figure 2. The reference architecture.

The user pilot system

The pilot system used in the OBOE project was a subset of the Introspection system developed at OTC for managing the marine seismic acquisition process. Introspection is the new family of tools that support improved planning, reporting and tracking of the marine seismic operation. The main vision of the system is:

- Provide ”adequate information at the finger-tip”:

- By easy access to updated "near real-time" daily activity-, project status- and revenue information by using the web intra-net
- By “processed” information direct to decision makers

- Provide "knowledge management":
  - By managing our historical data and thereby supporting improved decision-making for new bids and improved operation execution
  - By having an integrated solution that support and optimize the business processes

Figure 3. shows the high level architecture of the pilot system. The system also contained an access-right service and a name-service.

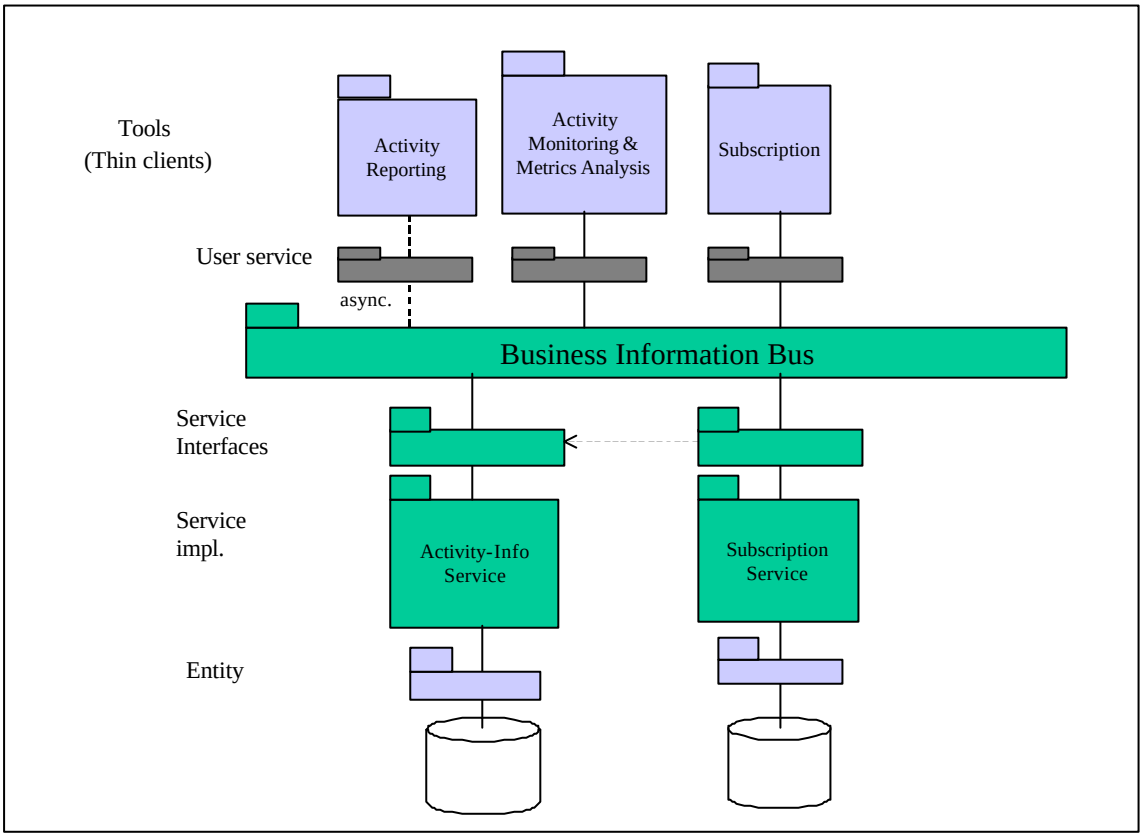

Figure 3. The main architecture of the pilot system

Figure 4. and 5. shows the Activity monitoring and metrics analysis tools. These were implemented as Java applets isolated from the rest of the infrastructure by the user-service layer. The service components on the server-side exported their interfaces as CORBA objects.

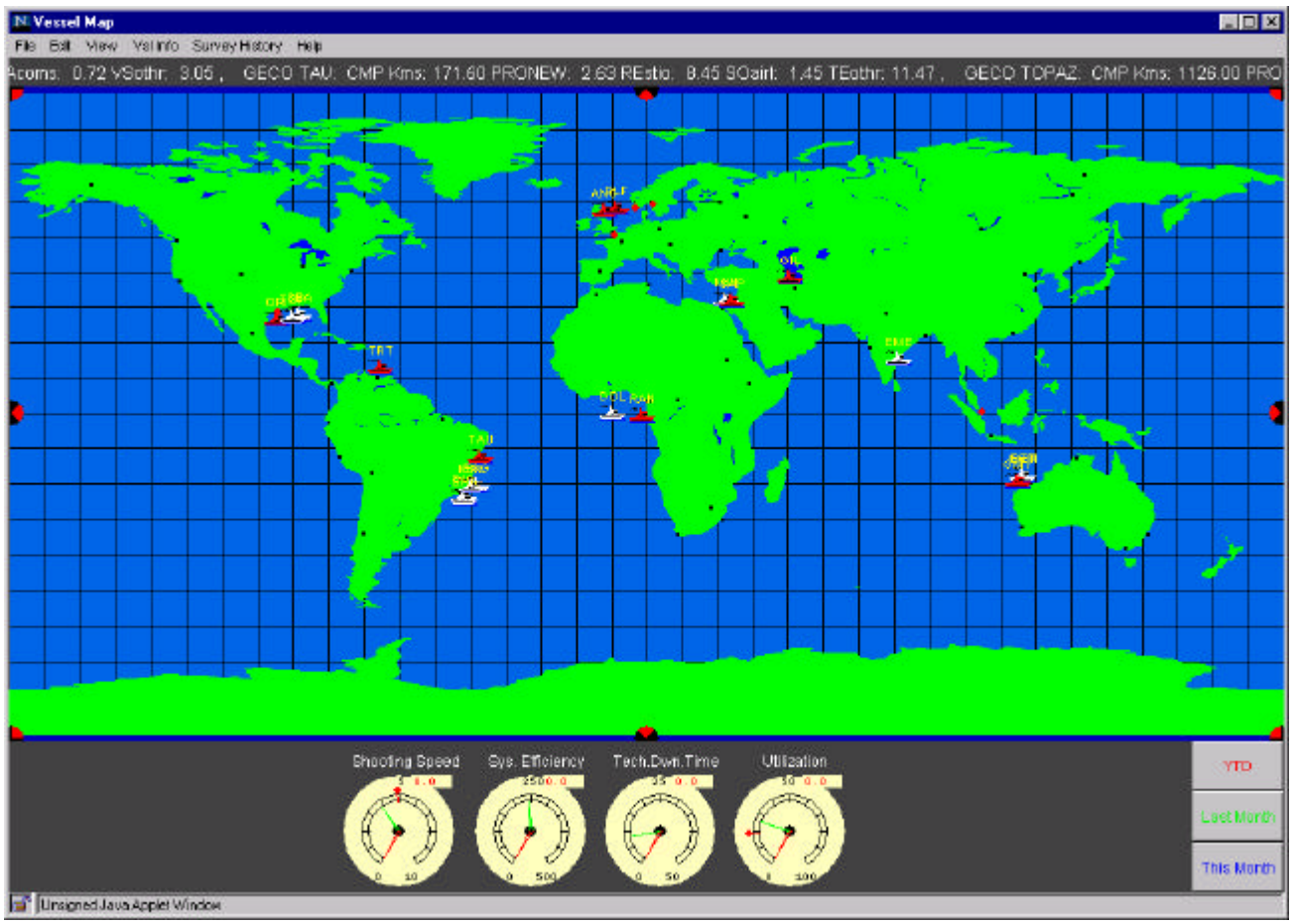

Figure 4. The interactive vessel-map tool

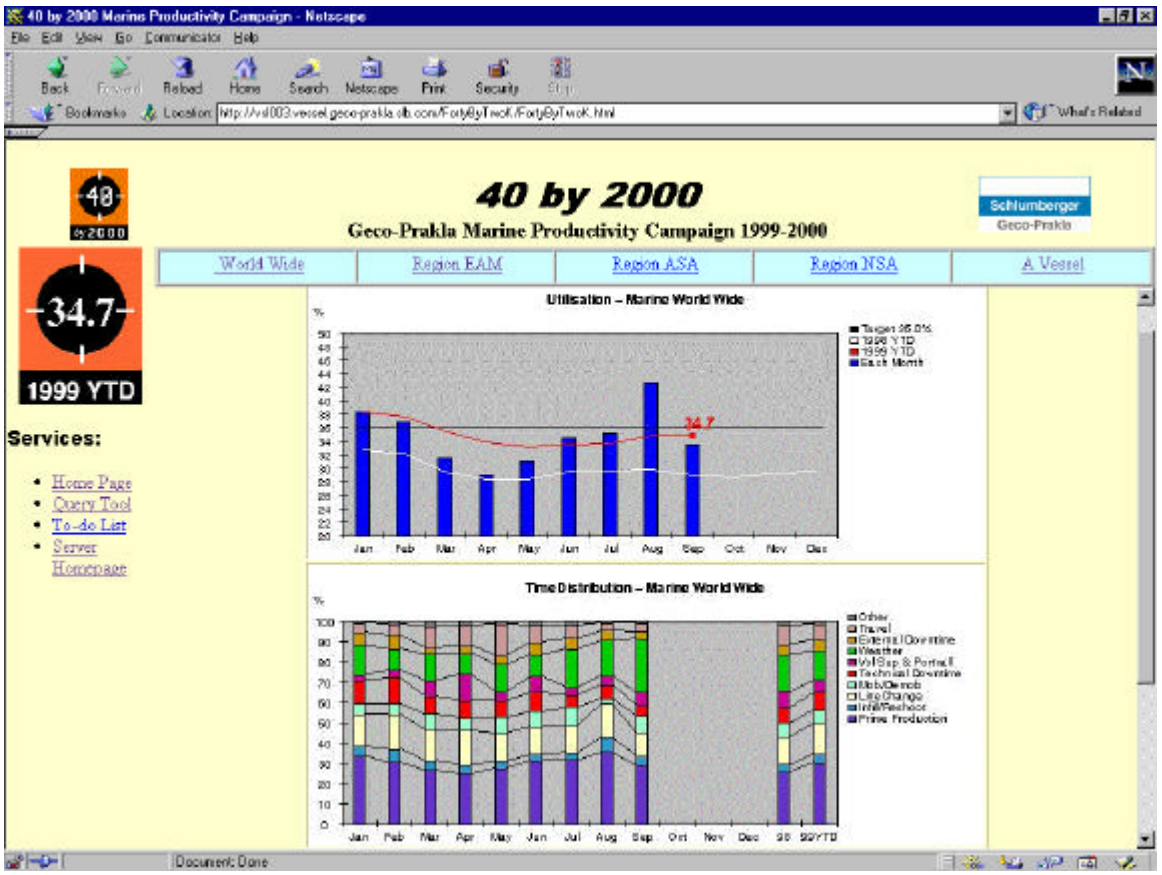

Figure 5. The utilisation and business metrics tool.

Meta-model mapping

The UML standard offers the possibility of making extensions to the UML meta-model. At the model level these extensions appear as stereotypes. By using this mechanism, desired concepts may be integrated into the UML models. Commercially available UML tools like Rational Rose that enable integration through API's or scripting, make it possible to perform model checking and specialized code generation to support specific needs. Thus, UML meta-model extensions facilitate a flexible way of supporting context, domain or architecture-driven concepts at the model level.

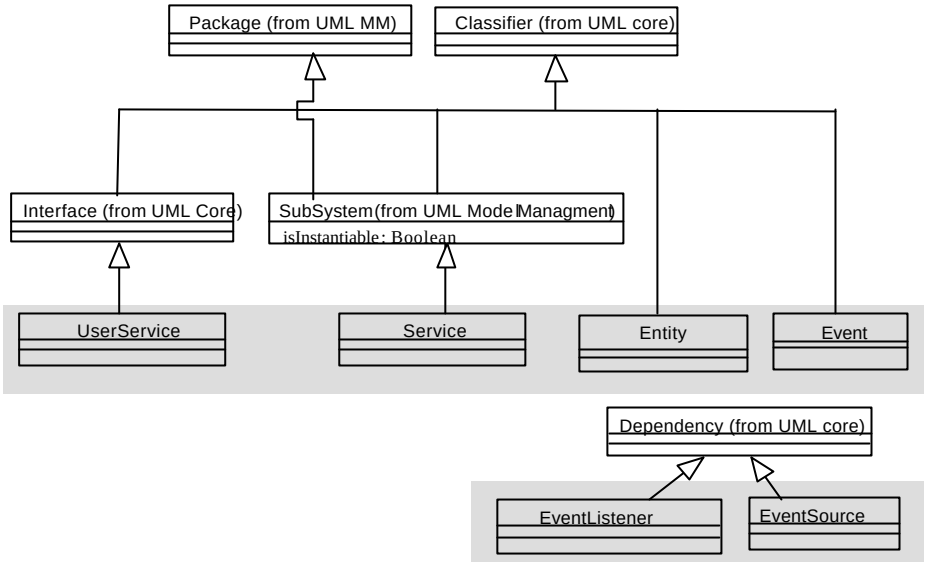

Figure 6. The meta-model

The metamodel defined in figure 6. extends the UML metamodel with concepts that correspond to the reference architecture. This includes the UserService, the Service and the Entity concepts. In addition, the metamodel supports an event model for handling business events. This includes the Event, the EventListenerDep and the EventSourceDep concepts. The new concepts are marked with gray background in figure 6.

The metamodel concepts

The UserService is a subclass of Interface from the UML core package. A UserService is only a collection of operations with a name.

The Service inherits SubSystem from UML Model Management. A Service logically contains a set of Entities, and serves as the controller of the Entity interactions. A Service component is instantiable and access transparent. Access transparent means that the Service components are registered and available on the net, so a Service component will be a CORBA object in a CORBA implementation. A restriction defined for the Service components is that they only may have relationships to Entities or to other Services. This is to ensure the described independence between the layers defined in the reference architecture. The following OCL[11] statement defines this formally:

```
self.allOppositeAssociationEnds -> forAll (a | a.type.oclIsTypeOf
     (Service) or a.type.oclIsTypeOf (Entity))
```

The Entity inherits Classifier from the UML core package. An Entity component is persistent and is access-transparent. A restriction defined for the Entity components is that they may only have relationships to other Entities. This again is to ensure that the Entity components are independent of the rest of the system. This is defined in OCL as follows:

```
self.allOppositeAssociationEnds -> forAll(a |
     a.type.oclIsTypeOf(Entity))
```

The metamodel supports an event model that includes the concepts *Event*, *EventListenerDep* and *EventSourceDep*. The event model is based on the JavaBeans event model. However, the event model has only an event hierarchy, not an event interface hierarchy like JavaBeans.

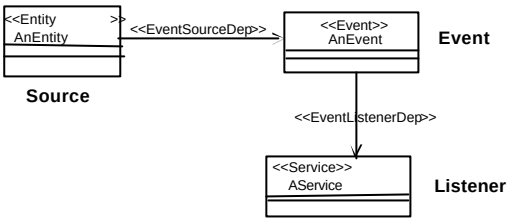

Figure 7. Event Model example

An event is represented as an object. An event can only have attributes. The only methods in an event object is the constructor and operations for the attributes. The attributes are read-only. An event is dependent of an event-source to be instantiated, and event listeners register themselves to be notified of events. Services and Entities may be event sources and event listeners, while UserService may register to be an event listener. The event model is illustrated with the example in figure 7.

The figure shows an Entity that generates events of type an Event and a Service that has registered to be a listener of that event.

The Component Modeling Language (CML)

The new concepts that were introduced in the distributed business system UML profile were inspired from the CORBA interface definitions, the Java-1.1 event model and the object database management group (ODMG) relationship, persistence and transaction models. In addition to the UML profile also a lexical language was developed as an extension to CORBA IDL. The language was called Component Modeling Language (CML) and supported a textual modeling notation as a complement to the graphical notation in the UML profile.

CML = interfaces (CORBA 2.0 IDL)
+ events (Java-1.1 event model)
+ relationship (ODMG)
+ persistence (ODMG)
+ transactions (ODMG)

The CML grammar is based on the CORBA IDL grammer and the syntax is similar to IDL.

The **methods** and **attributes** are semantically and syntactically the same as IDL, with the inclusion of the transient and factory keywords for attributes. A transient attribute have no persistent state and a factory attribute is shared between all instances of the same type.

The **event model** in CML is based on Java 1.1 event model. Both Service and Entities can subscribe and generate events. To export an event the **signal** keyword is used and to subscribe to an event the **subscribe** keyword is used. The event object is declared using the **event** keyword and can only have immutable attributes. Figure 8. shows an example of the CML declarations.

Relationships in CML conform semantically and syntactically to the relationship model in ODMG's ODL definition. A Service can have relationship to other Services and Entities. An Entity can only have relationship to other Entities. CML supports 4 types of relationship**: set, list, map and bag**. **Inverse relationship**is also supported and is identical to the ODMG standard.

An **Entity** is defined using the **entity** keyword. An entity has attributes, methods, relationship to other entities and supports event generations and subscriptions. See figure 8. for an example.

The **Services** correspond to the distributed components that can be located on the network and looked up through a naming service. The service keyword is used to define a service interface. Services includes the full relationship- and event-model.

```
module ActivityService
{
   service ActivityService
   {
       // Relationships
       relationship map< String, SurveyProject> SurveyProjects;  // Key eq VesselName
       relationship list< Vessel> Vessels;
       relationship set< Client> Clients;

       // Methods
       boolean login( String aUserName, String aPasswd);
       MarineActivity[] getActivityByCategory( String VesselName,
                                               String Category, TimeInterval Interval);
      SurveyProjects[] getSurveys( VesselList Vessels, Area anArea,
                                   ClientList Clients, TimeInterval ..);
       // Events
       signal NewActivity;
       signal MonthlyStatistics;
   }

   event NewActivity
   {
       attribute String ActivityType;
       attribute MarineActivity Activity;
   }

}

module SubscriptionService
{
   Service SubscriptionService
   {
       boolean login( String aUserName, String aPasswd);
       void addPublication( Publication aPublication);
       boolean addSubscription( String aName, Subscriber aSubScriber,
                                Publication aPublication);

       subscribe ActivityService::NewActivity;
       subscribe ActivityService::MonthlyStatistics;

       relationship map< String, Subscription> Subscriptions;
   }

   Entity Subscription
   {
       attribute String Name;
       attribute Subscriber aSubscriber;
       attribute Publication aPublication;
   }
}
```

Figure 8. A CML example from the pilot system

Using the Framework

A typical scenario of using the OBOE model driven development framework is depicted in figure 9. The startpoint is a UML modeling tool with a distributed business system UML-profile. This will alow one to create an UML model for a distributed business system including events relationships and transactions. Figure 10. shows such an UML model. The next step is to run a script that interprets the UML model with the profile extensions and generates the textual CML model. Most UML tools support scripting facilities to traverse the internal meta-model. The CML model is then parsed by a code-generator tool that outputs platform specific middleware code.

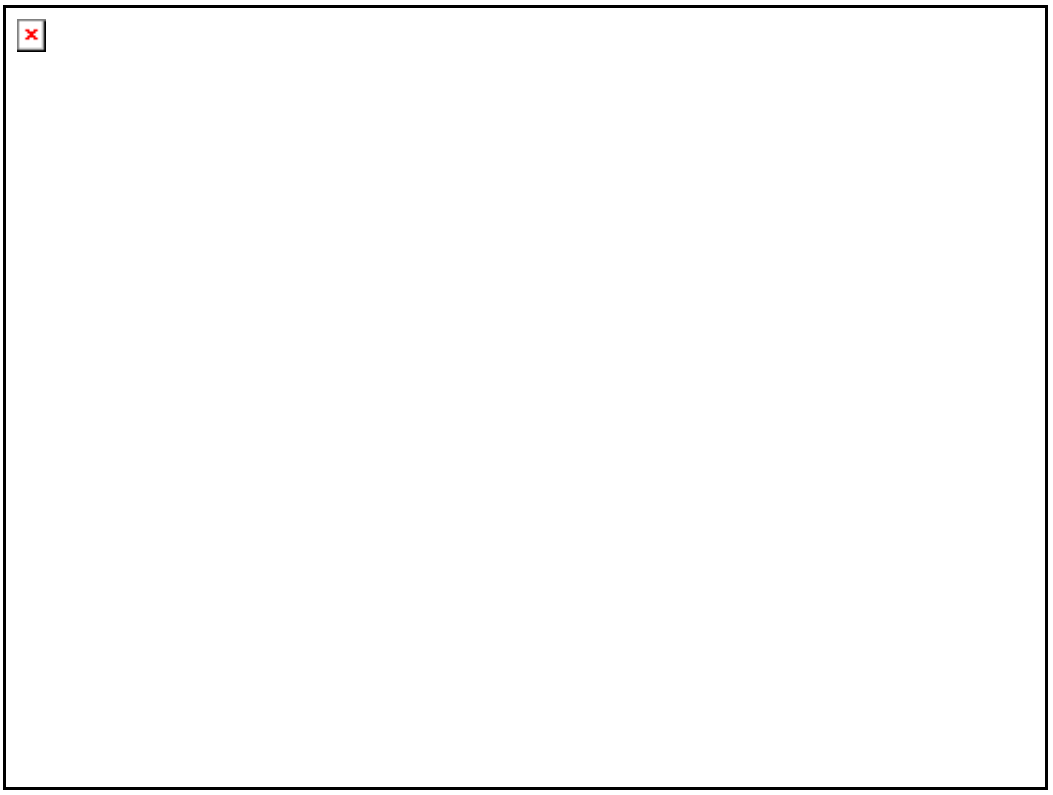

Figure 9. Activities and artefacts in a typical model driven development scenario.

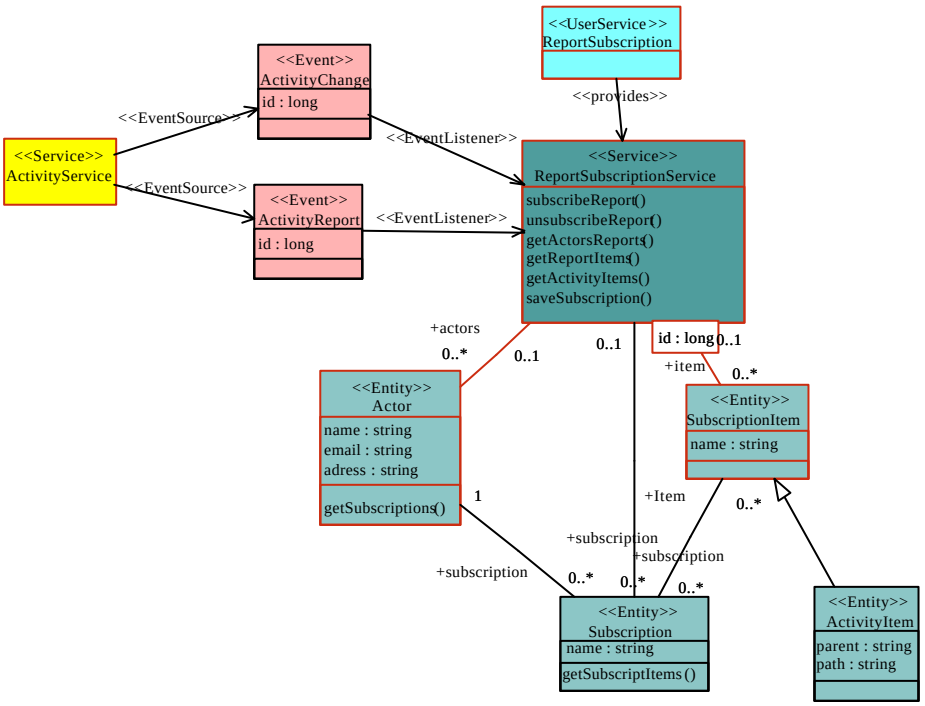

Figure 10. Report subscription Service. UML with Stereotypes.

Summary

This paper has described the way of doing Model Driven Development (MDD) as developed in the OBOE project. By applying formal models on all levels of system-analysis, there is a substantial potential for reuse at different levels of abstraction. This will make it easier to remodel a system to fit changes in the business needs. The separation of platform independent models from platform specific models improves the potential for faster development by supporting code-generation and communication middleware independence. MDD will also improve the quality of the system when it comes to performance, changeability and maintainability by using well defined reference architectures that are domain specific.

There are several promising initiatives going on in the industry that focus on MDD along with the OBOE project. OMG published a white-paper describing Model Driven Architecture (MDA)[12] late in the year 2000 and made a formal announcement of the initiative in March 2001. This is now a key focus area for OMG. There is also a new EU-founded project called COMBINE[13], where WesternGeco is one of the partners, that will continue the work started in OBOE.

References

1. M. Jackson, System Development, Prentice Hall, 1983
2. O. J. Dahl, B. Myhrhaug and K. Nygaard, Simula 67 Common Base Language, Norwegian Computing Center , Oslo 1970, NCC Publication S-52
3. UML Consortium, "UML Semantics" Rational Software Corporation Version 1.1, 1 September 1997.
4. OMG, *Object Management Architecture Guide*, Third ed: John Wiley & Sons, Inc., 1995.
5. J. Miller and J. Mukerji (eds), Model Driven Architecture (MDA), OMG, Document number ormsc/2001-07-01
6. M. Jackson, Problem Frames, Addison-Wesley, 2001
7. OBOE, "OBOE whitepaper" ESPRIT project no 23.233 revision 1.0, 1999.
8. A-J, Berre, B. Kvalheim, A. Solberg, J. Oldvik and B. Nordmoen, OBOE: a UML profile and lexical language for enterprise distributed computing, EC-workshop for GIS systems, 1999
9. A. Solberg, T. Neple, J. Oldevik and B. Kvalhem, A flexible framework for development of component-based distributed systems, SINTEF, fall 1999
10. R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, "The Object Database Standard: ODMG 2.0" in *The Morgan Kaufmann Series in Data Management Systems*, J. Gray, Ed. San Francisco: Morgan Kaufmann Publishers, 1997, pp. 288.
11. UML Consortium, "Object Constraint Language Specification" Object Management Group Version 1.1, 1 September 1997.
12. R. Soley (eds), Model Driven Architecture white paper, November 27, 2000
13. A-J. Berre (eds), COMBINE Green Paper, EU project number IST-1999-20893, September 2001