Business Modelling for large and Enterprise Systems and its Integration with UML 2.0

1. Introduction and Objectives

The aim of this paper is to show how large systems are analysed and designed using a number of standard techniques from the business modelling literature (Rummler and Brache 1995), UML and the patterns community (see Gamma 1994, Buschmann 1996). The approach is holistic in the sense that we use standard and proven technology from various sources to define a process that allows us to map customer wants and needs to design blueprints and patterns. This is the so-called Datasim Development Process (DDP) and is based on the author's work in the object-oriented field. The major characteristics and defining features of DDP are that it attempts to provide a seamless path between the different phases of the software development lifecycle and it uses proven and documented expertise whenever possible. For example, the process makes use of the following techniques:

- Business process modelling and activity diagrams
- Viewpoint-driven requirements analysis (see Sommerville 1997)
- UML class diagrams, sequence diagrams, statecharts and use cases
- Architectural patterns (see Buschmann 1996)
- UML analysis classes (Boundary, Entity and Control classes)
- Creating design blueprints from UML class diagrams

We show in later sections how these techniques are related and how they are integrated into DDP. There are many books that deal with some of the above techniques but to my knowledge there are none that deal with all of them. Furthermore, most of the object-oriented literature concerns itself with analysing and designing specific applications that are modelled using specific classes and specific interactions. A few examples from the literature come to mind:

- Home Heating System
- Automated Teller Machine (ATM)
- Elevator Control System
- Warehouse Management System
- Graphics Compiler Systems
- Resource Monitoring Systems

While it is a laudable aim to model toy problems they do not always help the analyst when he wants to solve his own specific problem. For example, most textbooks give no guidance on how to 'morph' an ATM application to similar applications such as drink vending machines, gambling machines and systems that offer access control to resources in a computer network. We wish to break away from the 'one-off' systems and instead we concentrate on looking at families of applications that have similar behaviour, functionality and structure. To this end, we introduce the concept of a Domain Architecture in this paper. We shall see that a domain architecture is a meta model in the UML sense. It should be used as a template for instantiating specific applications. For example, the Access Control System (ACS) is a domain architecture that includes ATM, drink vending machine and applications where access control, authorisation and authentication play a major role. In other words, if you know how to analyse an ATM application this knowledge will help you when you model a drink vending machine application because the two applications are similar in many ways. This remark will be sharpened in later sections.

This paper is structured as follows: in section 2 we describe the full software lifecycle that starts with a so-called Customer Requirements Specification (CRS) for a system and ends when we document design blueprints for it. Section 3 gives an introduction to domain architectures and describes the five major types that can be used to describe many applications occurring in the literature and in real-life applications. These architectures are meta models and can be used to instantiate and generate specific applications. In particular, we adopt the Meta Object Facility (MOF) specification because it provides a framework that allows the analyst to work on different levels of abstraction. Section 4 gives a brief introduction to the defining characteristics of Domain Architectures. Section 5 discusses some application areas for meta models and domain architectures.
A detailed treatment of models and metamodels for object-oriented system development can be found in [Duffy].

2. The DDP Software Lifecycle

Our main objective in this paper is to describe how enterprise systems are modelled using UML and related standards such as the Meta Object Facility (MOF). In particular, we are interested in modelling a system from the moment the idea is conceived to develop it to when the system has been delivered to the customer (and even after that point). In order to reduce the scope for the moment we consider the so-called software lifecycle model as shown in the UML activity diagram in figure 1. In this diagram the ellipses represent large-grained activities that produce output from input (represented by rectangles). This lifecycle has three main phases:

I: *Requirements Determination*. The phase where we discover the major viewpoints (perspectives) and requirements in the system. A viewpoint is a description of how groups of stakeholders view a system or what they expect from a system. We then 'decompose' a viewpoint into a number of requirements that realise that viewpoint. Persons, groups of persons and systems are responsible for the execution of these requirements and are called stakeholders. The process in this phase is top-down and this allows us to trace all entities in the system, in particular their dependencies on other entities.

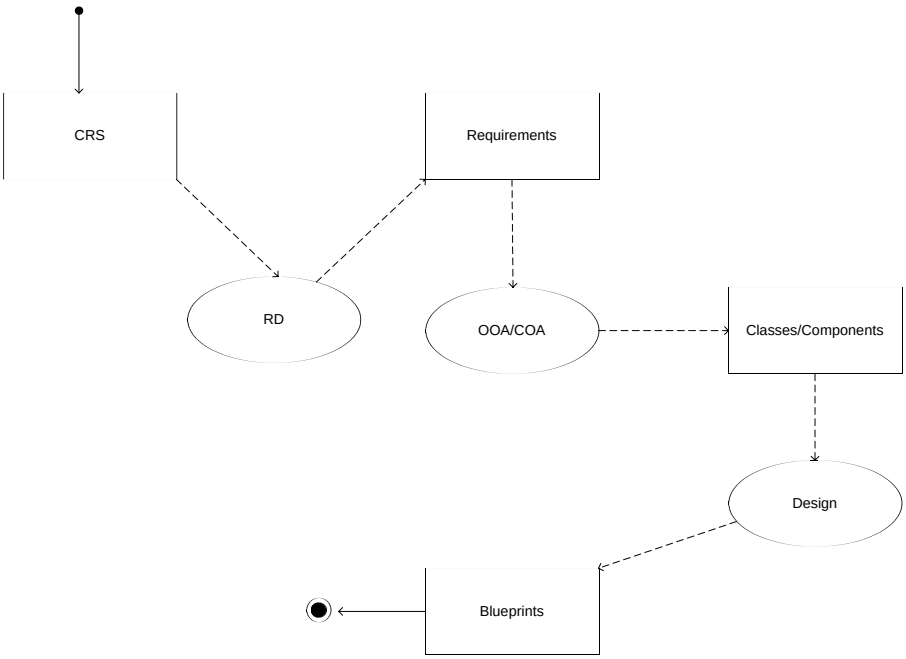

Figure 1 Software Lifecycle

II: *Analysis*. In this phase we discover the modules (for example, objects and components) that realise the requirements from phase I. In particular, we describe how the modules are incorporated into the overall system architecture, how they are related to each other and what their interfaces and attributes are. This is the phase where UML syntax techniques come into play. We distinguish between object-oriented analysis (OOA) and component oriented analysis (COA) ®. The distinction lies in the fact that objects have a single interface while components may realise many different interfaces. A specialisation of COA occurs when we model real-time and process-control applications by the UML Real-Time extensions (see [Selic98]). DDP includes provisions for RT UML integration. In fact, the architectural heuristics in RT UML (and its ROOM ancestor) have been integrated into a number of DDP domain models.
We apply the Presentation-Abstraction-Control (PAC) (see [Buschmann96], [Bass98]) because we have seen in real applications that it is highly scalable and is suitable for large heterogeneous, distributed systems. It also subsumes the well-known Model-View-Controller (MVC), Client/Server (C/S) and Jacobson Boundary-Entity-Control (BEC) patterns (see [Jacobson99], page 71 and [Buschmann96]). Furthermore, it acts as a 'supporting frame' for the entities in UML. In this case we are able to define a process for OOA and COA by integrating the UML syntax elements with PAC. It is surprising that UML has not (yet) accepted PAC and incorporated it into the language.

III: *Design*. This phase processes the products from the analysis phase and it attempts to harmonise these products with the software requirements in a system. The basic premise in this phase is that good software must adapt to accommodate changing requirements and we thus take account of volatile software and hardware environments (both physical and organisational). To this end, the process in design attempts to locate these volatile entities and to devise schemes to allow designers to create flexible and reusable software. Enter design and architectural patterns (see [Gamma94], [Buschmann96], [Schmidt00])! These patterns promote designer effectiveness by offering guidelines on how to achieve desired levels of flexibility.

An optional step during design is to document all design decisions in a quantitative manner. To this end we apply, the Quality Function Deployment (QFD) method to map high-level software requirements to lower-level ones. This technique has been used for many years in the automobile industry (see [Cohen95]) but very little has been done to apply it to software development. A notable exception is [Shaw96]. Most design processes are creative ('we do not know how long it will take to develop the system but it will be very beautiful when it is finished') while QFD always keeps customer wishes on the radar screen. QFD leads to what we call 'routine design', a process that is well developed in other disciplines such as mechanical engineering and chemistry.

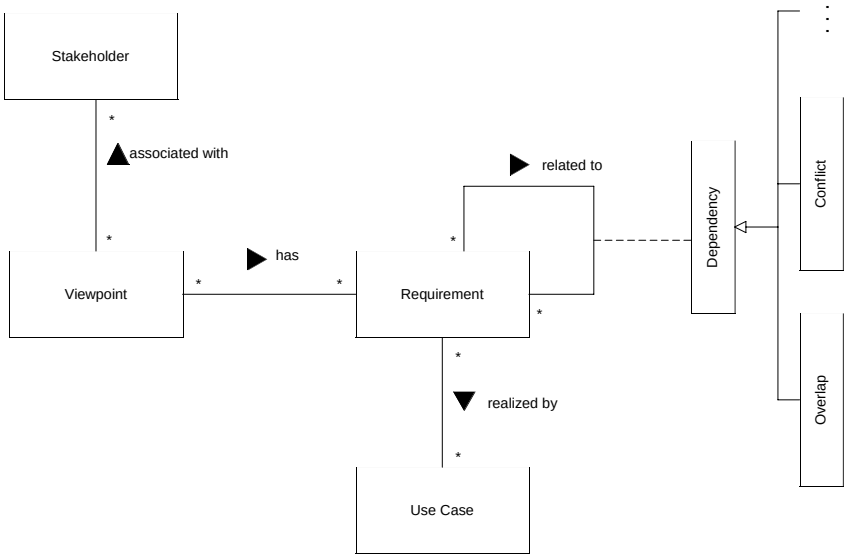

Figure 2 Artifacts from the Requirements Determination (RD) phase

Figure 1 is a ‘black box’ description of the work flow in the system development process. It is silent on a number of issues that demand our attention in this paper and elsewhere. It does not address the following issues. First of all, the structure of the products of each activity has not yet been described. By structure we mean the entities, their attributes and the inter-entity relationships. Second, we must define a step-by-step process that describes how an activity’s output is produced from its input. Finally, we must be sure on what we are modelling, whether it is a specific system (for example, an ATM application) or a class of applications sharing similar characteristics. This last issue receives our attention in this paper. We are unable to deal with the first and second issues in great detail here. However, we do give an example of the structure of the entities from the RD activity and the details of the actions in the OOA activity. The entities from the RD activity are shown in figures 2 and 3. Figure 2 shows the entities that have to do with behaviour and requirements. In particular, we model the relationships between the different requirements and the relationships between requirements entities and use case entities. Figure 3 is an abstract description of a context diagram. We paraphrase it by saying that the system under discussion (SUD) consists of subsystems and furthermore the SUD is related to its external actor systems in some way.

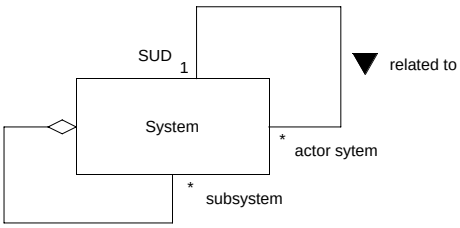

Figure 3 System Context Model

Figure 4 depicts the internal actions and subactivities within the OOA activity. We note that the input consists of use cases on the one hand and a system context on the other hand. The objective is to produce a consistent set of classes including their attributes, operations and the corresponding structural relationships with other classes. Looking at figure 4 we see that there are two main ‘flows’, namely the behavioural flow and the structural flow. The structural flow processes the system context and transforms it into a lower-level class architecture consisting of UML analysis classes (Boundary, Entity and Control classes). This is achieved in steps 1 and 3 in the diagram. The behavioural flow maps each use case to a sequence diagram (step 2). Once all sequence diagrams have been created we are then in a position to collate all messages that are related to each class that was discovered in the structural flow (step 4). Step 5 is concerned with the creation of state charts for the discovered classes. Finally, step 6 integrates the products from the behavioural and structural flows.

There are a number of benefits and advantages if we view software development as a lifecycle model such as is shown in figure 1:

- Separation of concerns: there are clear dividing lines between requirements determination, analysis and design phases. Each activity displays a high degree of internal cohesion and exhibits loose coupling with other activities. For example, only the use cases from figure 2 are needed by the Analysis activity.
- We can model all the artefacts in figures 1, 2, 3 and 4 by UML models. In fact, we shall model them using the new MOF specification. This opens up the possibility for data repositories and coupling with other models.
- Standardisation. Each activity is described and executed using standard procedures, the products are documented using standard templates and communication between developers is improved because they are using the same vocabulary and work practices.
- The model allows us to integrate UML with business processes on the other hand and with design and architectural patterns, on the other hand. This is highly desirable and necessary if we wish to use UML in real-life applications and situations.

We now introduce the concepts of modelling and meta modelling for enterprise systems.

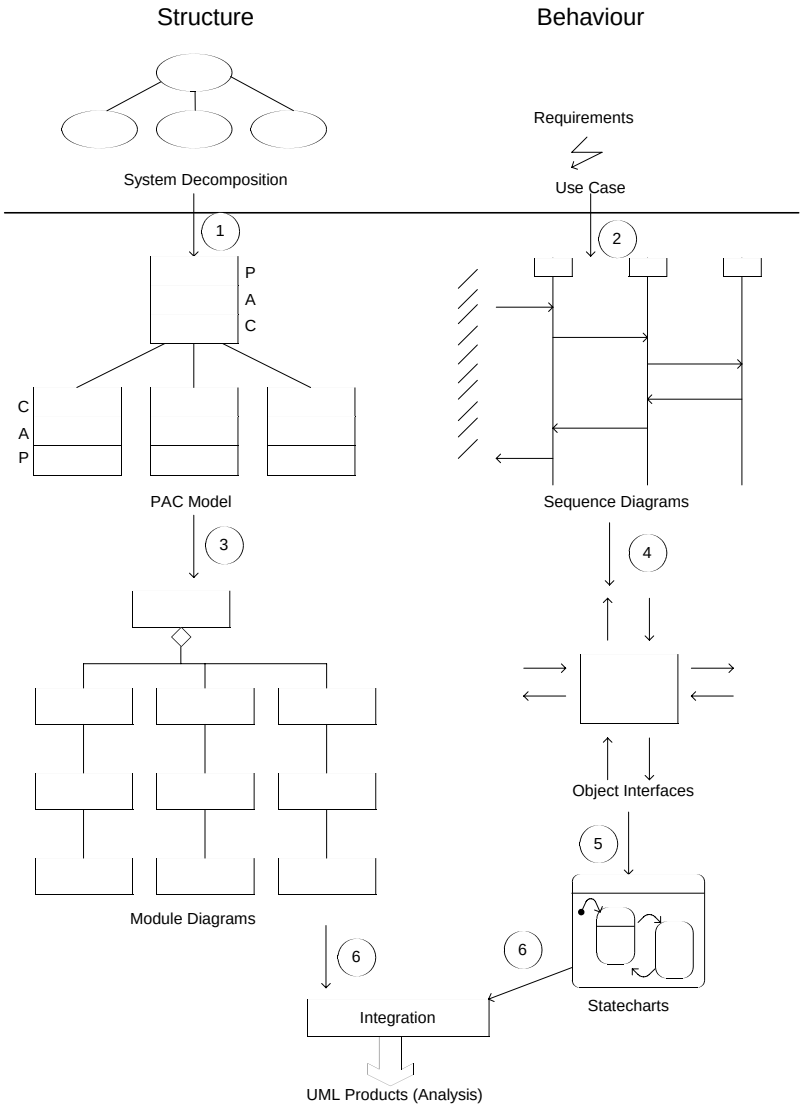

Figure 4 OOA Process (UML)

3. An Introduction to Domain Architectures and Meta Models in UML

There are several attention areas that must be addressed when developing software systems. In particular, there are numerous viewpoints or perspectives that can be taken. Each viewpoint corresponds to a so-called dimension. For example, the lifecycle phase in figure 1 is a dimension having the values 'Requirements Determination (RD)', Analysis (OOA/COA)' and 'Design'. We wish to model other dimensions and in general we can model any software problem as a point in a four-dimensional space as shown in figure 5. The dimensions are:

- Lifecycle phase (as already discussed in section 2)
- Viewpoint (this is a well-accepted approach in many software methodologies)
- Meta level (are we looking at a specific system or a family of systems?)
- Quality dimension (for example, are we focusing on functionality, portability…?)

This analogy with Cartesian geometry is very useful because we can describe any software activity as a point instance in this 'software' space. We must agree on the ordering of the dimensions; we state that a point instantiates the array (Meta Level, Viewpoint, Lifecycle Phase, ISO 9126 Quality). For example, let us suppose that we are carrying out an analysis (OOA) of the ATM application and that we are interested in the portability aspects of the Entity classes in the class architecture. The corresponding point instance will then be (M1, Structural, Analysis, Portability).

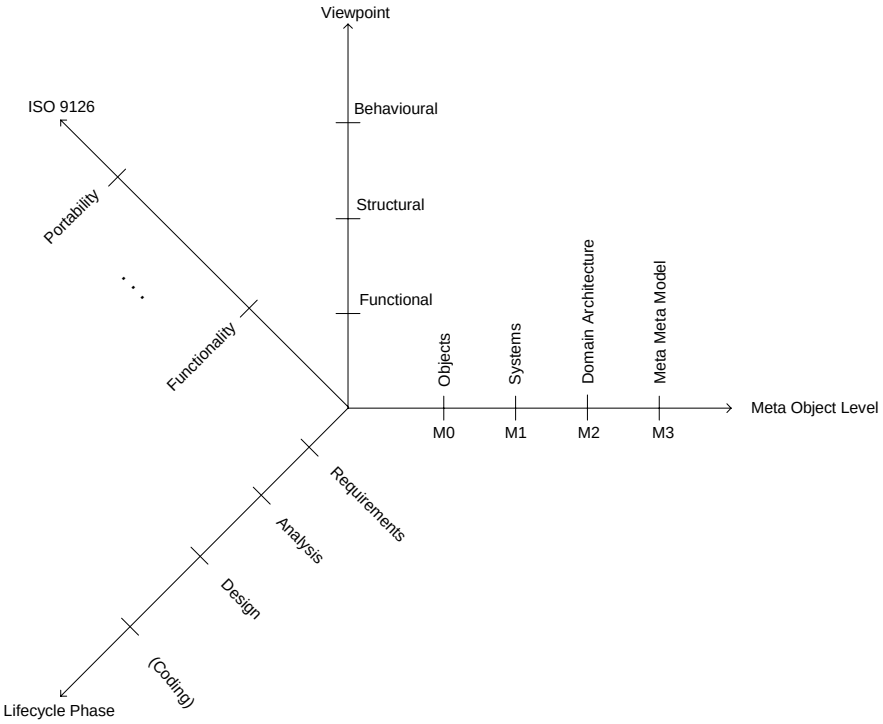

Figure 5 Dimensions in Software Development

We now introduce and discuss the Meta Object level dimension in figure 5. This is based on the OMG's Meta Object Facility (MOF) specification which we now introduce. The MOF is an object-oriented modelling language that allows us to describe information and data at any level of abstraction. OMG has produced a four-layer scheme in MOF as shown in figure 6. The names of the different layers can be confusing and this is why we prefer to use numbers that indicates relative positions in the hierarchy. The four layers are:

- M0: (user object layer) contains the data and information that we wish to describe
- M1: (model layer) the meta-data that describes information from level M0
- M2: (meta-model layer) describes the structure and semantics of meta-data
- M3: (meta-meta data layer) describes the structure and semantics of the meta-meta-data

Our interest in this paper is in examining the layers M1 and M2 and the relationships between them. In principle, each value corresponding to the dimensions in figure 5 can be modelled as a layered

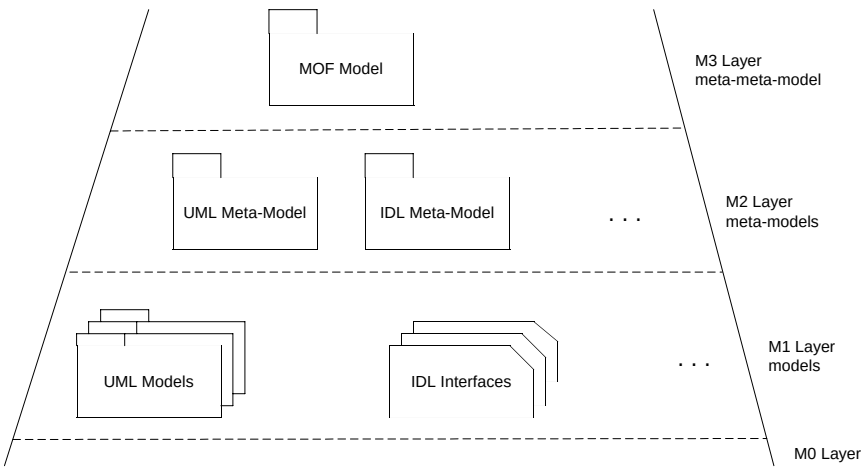

Figure 6:MOF Meta-data Architecture

regime. For example, we could carry out an analysis (OOA) of layer M2 or layer M1; of course, this implies that we must examine the corresponding Viewpoint and ISO 9126 Quality dimensions.
We give an example of a MOF specification for a simple ATM application. We concentrate on structure, in particular the class Account. This models a customer's current account. The layers contain the following types of information:

- M0: Account ("Daniel Duffy", 123456789)
- M1: Account type having two fields ('Person' and 'ACC#'), each having a name and a value
- M2: Defines what it means to be an Account type. In this case we model it by meta attributes and each one is described by meta attributes for its name and type
- M3: this is usually hard-wired.

Our long-term objectives are to model each entity in the software lifecycle and position it in its appropriate place in the MOF hierarchy. For example, the entities in figure 7 that are produced during the Requirements, Analysis and Design phases. These are examples of meta-model entities.

Viewpoint	Requirements	Analysis	Design
Functional	Goals/Processes/ Activities	Entity Model	Persistent Data (Creational Patterns)
Structure	Context Diagram Key & Satellite Systems	Classes in UML	Extended UML Classes (Structural)
Behaviour	Viewpoints Stakeholders Requirements	Use Cases Sequence Diagrams Statecharts	Behavioural Patterns

Figure 7: Products in Software Development

4. The Dimensions of Domain Architectures

The author has discovered and documented a number of so-called design categories or domain architectural types. A domain architecture is a description of a family of applications with each application having similar structural and behavioural properties. All applications in a given category deal with the same issues except that a given application in a category will has its own specific jargon. The main categories at the moment of writing are:

- Manufacturing (MAN)
- Resource Allocation and Tracking (RAT)
- Management Information Systems (MIS)
- Process Control and Real-Time (PCS)
- Interactive and Access Control Systems (ACS)

A given category is a description of a class of applications sharing similar structure and behaviour. In particular, all applications in a given category have similar Boundary, Entity and Control objects (see [Jacobson99]) and we are thus able to 'predict' to a certain extent what these objects will be for an application. Furthermore, it is possible to define and discover standardised viewpoints, requirements and use cases for a given category. The author has documented these objects and use cases and will

be discussed elsewhere. In short, high levels of reusability are achieved, not so much at the implementation level but in the analysis and design phases.

5. Application Areas for Domain Architectures

We give a summary of some subclasses and instances of the domain categories. The basic classification system is still evolving.
We describe some special applications that the author has been involved in:

Manufacturing Systems (MAN)
- CAD systems (creation of technical drawings)
- Visualisation (e.g. holographic images)
- Compiler (generate API interfaces from ASCII input)

Resource Allocation and Tracking (RAT)
- 'Classic' call-handling systems
- Order entry and processing systems
- Elevator systems
- Plastics extrusion systems
- The phases in software lifecycle ('metamodels')

Management Information Systems (MIS)
- Resource usage systems (e.g. project management)
- Portfolio (financial) monitoring systems
- Management of other applications (e.g. PCS)

Process Control and Real-Time (PCS)
- Environment monitoring and control systems
- Barrier option modelling (in risk management)
- Exception management for other systems (e.g. MIS)

Interactive and Access Control Systems (ACS)
- Drink vending machines
- Automated teller machines
- Interactive graphics

Finally, many of the 'toy' applications found in the literature are just instances of one or more of the above categories. We prefer to examine a given application from a more general perspective.
We finish this section with an example of how to model a system's context using the MOF model. We recall that a system's context depicts the relationship between the system and its external actor systems. The M3 model for this has already been shown in figure 3. We now look at the context diagram for a Resource Allocation and Tracking (RAT) domain as in figure 8. Here we see that there are six external actor systems each of which having well-defined responsibilities. This is a context diagram at level M2. Finally, we look at a system for registering customer orders, allocating resources and notifying customers of the status of orders. Its context diagram (at level M1) is given in figure 9.

6. Conclusions

We have given an overview of some issues that confront developers when analysing and designing large systems. The complexity of the undertaking is reflected in the different viewpoints that need to be taken if we wish to fully comprehend how to proceed. Once we know what has to be done we need to define a process. The products and artefacts need to be documented as models and metamodels and to this end we decide to use the OMG's Meta Object Facility (MOF) specification.

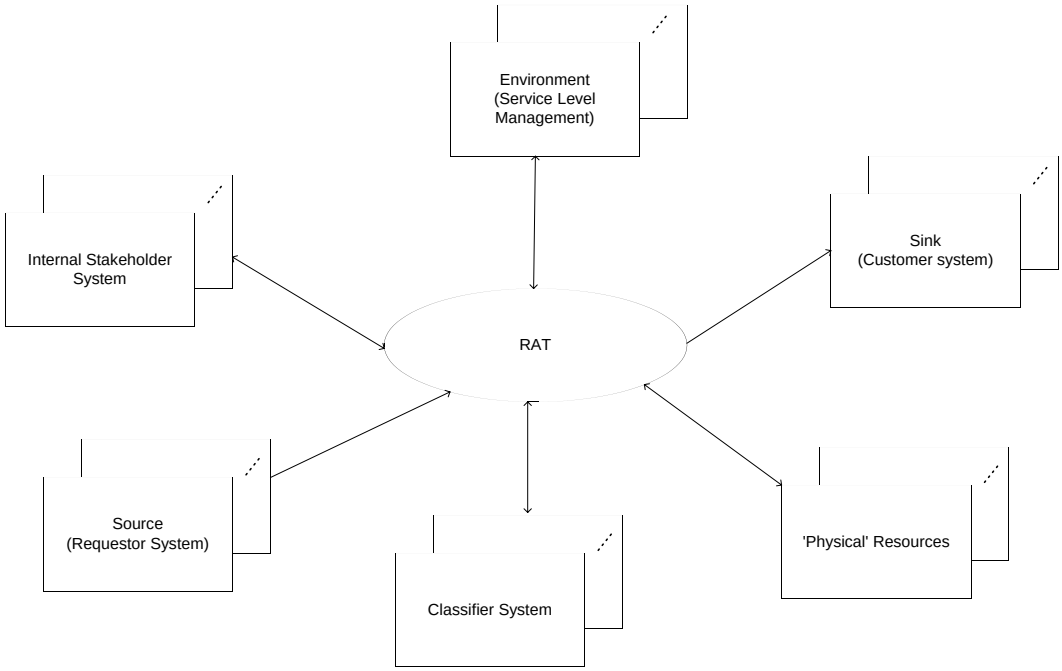

Figure 8:RAT Context Environment

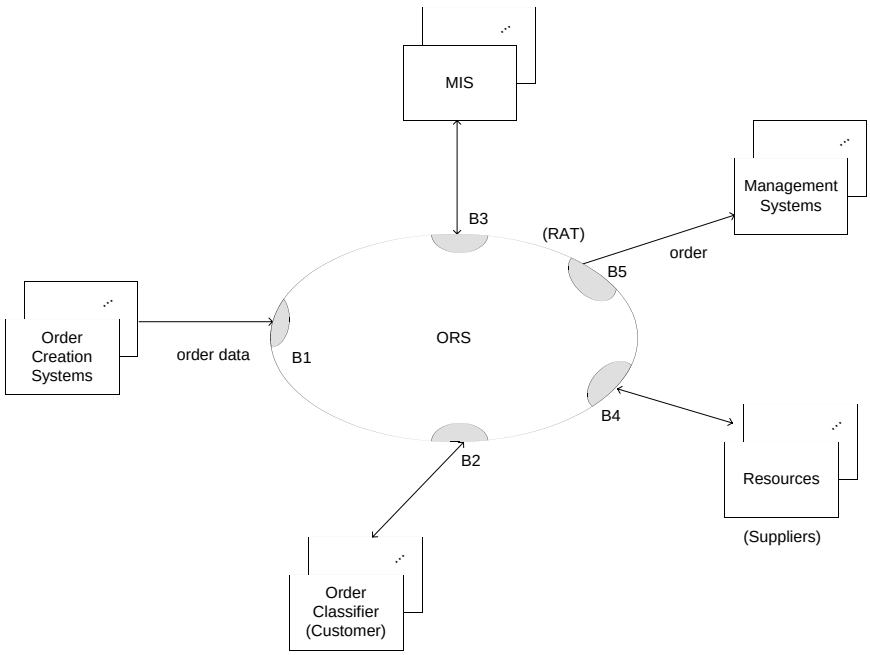

Figure 9 Context Diagram Order Scheduling System

We have introduced the concept of a domain architecture and some examples from several application areas. Domain architectures fit in the MOF M2 layer because they are templates for instantiating specific applications (alternatively, a specific application is an instance of a domain architecture) and applying them leads to major productivity and reusability gains.

REFERENCES

[Bass98] L. Bass, P. Clemens, R. Kazman '*Software Architecture in Practice*', Addison-Wesley Reading MA.

[Buschmann96] F. Buschmann et al '*Pattern-Oriented Software Architecture*' John Wiley & Sons Chichester 1996.

[Cohen95] L. Cohen '*Quality Function Deployment*', Addison-Wesley, Reading MA 1995.

[Datasim2000] Datasim Education BV '*Course on Requirements Determination*' 2000.

[Duffy] Daniel J. Duffy '*Domain Architectures: Models and Architectures for UML Applications*' John Wiley (to appear)

[Gamma94] E. Gamma et al '*Design Patterns*' Addison-Wesley Reading MA 1994.

[Jackson95] M. Jackson '*Software Requirements & Specifications, a lexicon of practice, principles and prejudices*' (ACM Press) Addison-Wesley Wokingham England 1995.

[Jacobson99] I. Jacobson et al '*The Unified Software Development Process*' Addison-Wesley Reading MA 1999.

[Kitchenham96] B. Kitchenham, S. L. Pfleeger '*Software Quality: The Elusive Target*', IEEE Software January 1996.

[Martin92] J. Martin, J. Odell '*Object Oriented Analysis and Design*', Prentice Hall Englewood Cliffs New Jersey 1992.

[Novak84] J. D. Novak, D. B. Gowin '*Learning how to learn*', Cambridge University Press 1984.

[OMG2000] Unified Modeling Language Specification v. 2.0 (Request for Proposal) Object Management Group 2000 (see www.omg.org).

[Rummler95] G. A. Rummler, A. P. Brache '*Improving Performance, second edition*' Jossey-Bass San Francisco 1995.

[Schmidt00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann '*Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*', John Wiley & Sons, Ltd. Chichester 2000.

[Selic98] B. Selic, J. Rumbaugh '*Using UML for Modeling Complex Real-Time Systems*', Rational Software Corporation 1998.

[Shaw96] M. Shaw, D. Garlan '*Software Architecture, Perspectives on an Emerging Discipline*' Prentice-Hall, Upper Saddle River, New Jersey 1996.

[Sommerville97] I. Sommerville, P. Sawyer '*Requirements Engineering*' John Wiley & Sons Chichester 1997.

[Swift98] Jonathan Swift '*Gulliver's Travels*' Oxford University Press 1998.

[Tingey97] M. O. Tingey '*ISO 9000, Malcolm Baldridge and the SEI CMM for Software*' Prentice Hall Upper Saddle River New Jersey 1997.

[Winston84] P. H. Winston '*Artificial Intelligence, second edition*' Addison-Wesley Reading MA 1984.

[Yourdon89] E. Yourdon '*Modern Structured Analysis*' Prentice-Hall Englewood Cliffs NJ 1989.