

Datasim Development Process (DDP) ®

White Paper

© Datasim Education BV
Schipluidenlaan 4
1062 HE Amsterdam
The Netherlands
www.datasim-education.com

Summary

This white paper is an overview of the Datasim Development Process that describes the software lifecycle starting from initial customer goals and business processes and ending with design and architectural patterns. The process integrates standard methodologies such as domain architectures, UML, patterns, objects and components. The main objective of this article is to show how DDP provides a defined process that supports both forward and backward traceability in the software lifecycle.

The DDP developed in the 1990's when the author and his colleagues were involved with the introduction of object-oriented and component technologies in small, medium and large organisations. Based on those experiences we have developed a process that we feel resolves a number of the shortcomings of current software technologies and that provides an environment in which different stakeholder groups can communicate about system development.

Keywords: Goals, core processes, viewpoints, stakeholders, requirements, use cases, UML, domain categories, routine design, design and architectural patterns, phase, business processing, requirements determination, object oriented and component analysis, object design

1. Introduction and Background

The origins of the Datasim Development Process can be traced to the late 1980's when our company Datasim became involved with object-oriented technology, especially in the area of C++ class libraries for Computer Aided Design (CAD) applications. There were high expectations about the possibilities of this new technology especially how it would allow developers to write software that was reusable, flexible and portable. In fact, it was touted as the next silver bullet. Those advocating the new technology believed that it was sufficient to find the objects in an application in order to write flexible software. For this reason, structured techniques and top-down process/functional decomposition techniques fell into disrepute.

The object-oriented approach began to gain wide acceptance in the late 1990's. However, the promise of reusability and flexibility has not been realised in general. The reason for this apparent failure is in my opinion, not a limitation of the technology but due to the fact that IT organisations were not mature enough to adopt and adapt OOT. In the late 1990's component technology started to emerge as a leading contender for the minds and hearts of IT professionals. The acceptance of EJB and COM/DCOM is proof that component technology is here to stay. Furthermore, agent technology is beginning to emerge as an 'extended' object-oriented technology. In fact, the OMG has a special interest group that is adapting UML to this technology and a number of OO researchers and consultants (for example, Mr. James Odell) are involved.

This article is structured as follows: In section 2 we discuss the strengths and weaknesses of current software technologies (in particular object and component technology) and we discuss some of the attention points that DDP addresses. In a sense we wish to adopt an evolutionary approach to software development by engendering a continuous improvement mindset. Section 3 is an executive overview of the DDP process, including a summary of its four main phases. We discuss each phase in some detail including the information flow through each one, which stakeholders are involved in each phase and how each phase is described and documented. Section 4 deals with the different phases in more detail. It includes a discussion on how the information flow is processed as a sequence of activities and how the deliverables are documented. We use standard techniques whenever possible as it is our aim to develop as little as possible of our own notation especially if there is an already accepted standard. What is different however is

how we approach software development and we believe that we have developed a defined process that can be applied to many different classes of problems. Section 5 discusses the Design phase, how it relates to the other phases in the software lifecycle and how its products are produced. Particular attention will again be paid to how the deliverables are created and we are thus not satisfied with rehashing UML diagrams without telling how they came into existence. Section 6 deals with a discussion of a number of so-called domain categories. A domain category is a generic container for all applications that share similar requirements, processes and structure. An example of a domain category is MIS (Management Information System); all applications in this category have requirements for data input validation, data merging, consolidation and reporting. We describe how these categories fit into the DDP and why they are so useful.

Remark: this article does not contain information relating to project management issues, software economics, resource scheduling or implementation details. Although these are vital in any real-life situation we exclude them in this presentation. They will be discussed elsewhere.

2. A Critique of Current Approaches to Software Development

This section is a summary of the personal views of the author on the maturity of the object-oriented paradigm, how it has been used in the last ten years, what problems it has led to and how workarounds have been discovered that resolve a number of shortcomings of the paradigm. As somebody once said, prediction is extremely difficult, especially predicting the future. If we knew what the limitations of the object-oriented technology (OOT) were ten years ago we probably would not have made the same mistakes, only other ones!

A fundamental and sometimes implicit assumption underlying OOT was that if developers were able to discover the domain abstractions (that is objects) for their particular applications then success would be guaranteed. This resulted in the top-down decomposition approach (see [Yourdon89]) being abandoned in favour of more creative and ad-hoc approaches to system decomposition. Another problem was that many IT professionals (including the present author!) thought that OOT was the new silver bullet and that it would cure all software ills. A consequence of this mindset was that very little attention was paid in the early years to integration with legacy systems. Finally, there was much confusion as to what an object really was. This problem is by no means new. Going back some years, Jonathan Swift (see [Swift98]) describes a similar situation when he relates how Gulliver (the main character in his book) recounts his meeting with a group of professors from Laputa. These professors belonged to the School of Languages and their aim was to improve communication by eliminating all words except nouns:

' We sent next to the School of Languages, where three Professors sat in Consultation upon improving that of their own Country.

The first Project was to shorten Discourse by cutting Polysyllables into one, and leaving out Verbs and Participles; because in Reality all things imaginable are but Nouns.

...

However, many of the most Learned and Wise adhere to the new Scheme of expressing themselves by Things; which hath only this Inconvenience attending it; that if a Man's Business be very great, and of various Kinds, he must be obliged in Proportion to carry a greater Bundle of Things upon his Back, unless he can afford one or two strong Servants to attend him.'

Returning the 21st century, we see that the application of the object-oriented paradigm has led to a number of far-reaching consequences. First, OOT engenders a bottom-up approach to system development that eventually leads to monolithic applications that are difficult to modify and to extend. To make matters worse, developers discovered multiple inheritance in the late 1990's thereby allowing them to create classes that were dependent on lots of other classes! Second, the principle of encapsulation, even though a good goal to strive for forces developers to think about data in the early stages of software development and this is unfortunate because inter-system interfaces are more important than data. Finally, since OOT was technology-driven little emphasis was placed on management issues.

The aim of this white paper is to show how Datasim has resolved a number of the problems and shortcomings associated with OOT. In particular, we adopt a multi-paradigm and holistic approach to software development because we think that the current technologies (including UML) are too focused on syntax and solution domain issues. *We do not pretend that we have solved all problems nor do we advocate DDP as a silver bullet!* However, we are not happy with the current states of affairs in OO land, mainly because of the difficulty of applying the technology to enterprise systems.

We hope to add some input to the ongoing discussions concerning UML 2.0 and it is our view that a number of the problems and shortcomings of UML 1.3 have been resolved by the DDP. We summarise some of the concerns and problems with UML in [OMG2000]. I quote from that article:

- UML metamodel is large, semantics is uneven and little distinction is made between the different phases of the software lifecycle
- Little support for component technology, in particular good mappings to EJB, Corba and DCOM
- Inconsistent use of diagrams and plethora of overloading options
- Model management and scaling problems

It is not our intention to discuss these problems in this paper but many of them can be attributed to the single fact that UML has grown in a bottom-up fashion (it springs from OOT which is by default bottom-up). It is our opinion that taking a top-down viewpoint and shifting focus resolves many of these problems.

3. Executive View of DDP

DDP can be viewed as a so-called software lifecycle model by which we mean that customer wants and needs (which tend to be abstract) are mapped to lower-level and more tangible requirements. The objective is to develop an executable system that runs on some computer (or family of computers) and which implements all customer requirements. There is such a huge cognitive gap between what the customer says he wants and how this can be implemented in a language such as C++ or Java that we are forced to partition the lifecycle into a number of so-called phases. Each phase is responsible for a part of the processing of the lifecycle. In particular, each phase processes input information and produces output information. Furthermore, various stakeholders are involved in each phase. A given phase has its own vocabulary; this means that graphical and textual notation will be developed to describe what is happening in a phase and how the phase communicates with its 'predecessors' and 'successors'.

DDP concentrates on four major phases. Each phase processes one major input entity and delivers one major output entity. The idealised information flow is shown in Figure 1 where it is displayed as a so-called process map (see [Rummler95]). Each rectangle represents a process that is essentially an I/O black box. We shall see in sections 4 and 5 what actually happens inside each box, in particular what the main activities are in each box. The dashed lines in Figure 1 represent so-called 'swimlanes' to distinguish between the different organisational units that are involved in each phase (note that we do not discuss the Implementation phase in this article).

The phases are:

I: Business Modelling. This is the phase in which business goals and core processes are discovered (see [Rummler95], [Sommerville97]). The DDP software lifecycle begins when a Customer Requirement Specification (CRS) has been presented. The CRS contains a list of the major features that the customer wishes to see in a system. The approach taken in DDP is to determine what the scope of a system is, how the system is to be decomposed into loosely coupled

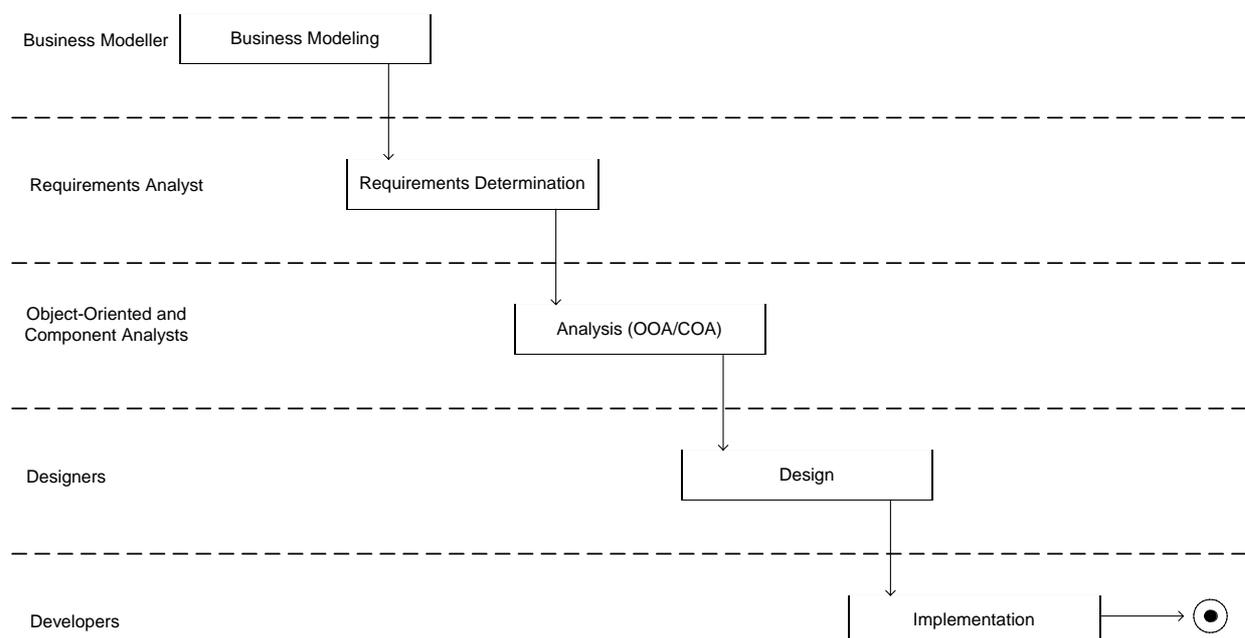


Figure 1 Process map for DDP

logical subsystems and (optionally) to determine which domain category the system falls under. The idea is to discover and document a stable structure that will be integrated with the deliverables from the Requirements Determination phase. Thus, our approach diverges from that taken in the standard literature in UML (see [Jacobson99]) where use cases are first discovered and documented and then the corresponding structure is elaborated. Our system decomposition process is top-down and is achieved by looking at the information flows in a system. In fact, business modelling and system decomposition activities should take place concurrently.

II: 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 for this reason. 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.

III: Analysis. In this phase we discover the modules (for example, objects and components) that realise the requirements from phase II. 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 application 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 subsumed 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. It would make life a lot easier for me if it came about.

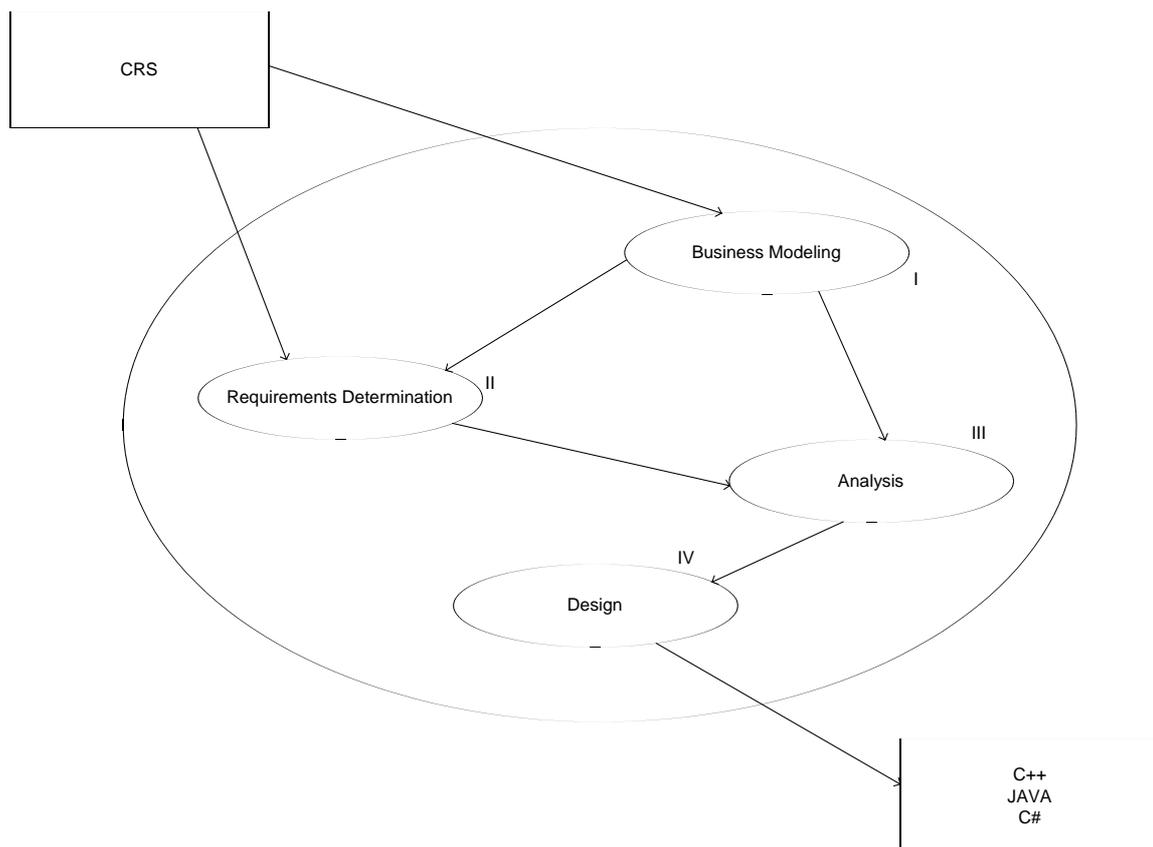


Figure 2 Systems in Lifecycle Model

IV: Design. This phase processes the deliverables from the analysis phase and it attempts to harmonise them with the software requirements in a system. The basic premise in this phase is that good software must adapt to accommodate changing circumstances and we must thus take account of volatile software and hardware environments (both physical and human). 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, the Quality Function Deployment (QFD) method is applied to map high-level software requirements to lower-level ones. This technique has been used for almost forty 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, chemistry and physics. As software designers, we have a lot of catching up to do with our colleagues in other disciplines.

Figure 2 is a representation of the different systems in the lifecycle model while figure 3 gives a summary of each phase including its information processing entities, which stakeholders are involved in the realisation of the deliverables, what kind of vocabulary is used and how activities, products and deliverables are documented.

	Input	Output	Stakeholders	Vocabulary	Graphical Techniques
Business Modelling	Customer Requirements specification (CRS)	Domain Architecture	<ul style="list-style-type: none"> • Domain Expert • Architect 	<ul style="list-style-type: none"> • Goals • Core processes • Activities • Domains 	<ul style="list-style-type: none"> • Process Maps • Activity Diagrams • System Decomposition • Domain Category
Requirements Determination	<ul style="list-style-type: none"> • Domain Architecture • CRS 	Requirements Description	<ul style="list-style-type: none"> • Requirements Analysis • Customer 	<ul style="list-style-type: none"> • Requirement and Requirements Attributes • Viewpoints • Stakeholders • Use Cases 	Templates that describe requirements
Analysis	Requirements Document (description)	Analysis Document	<ul style="list-style-type: none"> • Analyst • Requirements Analyst 	UML Modelling Concepts	<ul style="list-style-type: none"> • UML Class Diagrams • Sequence Diagrams • Statecharts
Design	Analysis Document	Extended Class Diagram Document	<ul style="list-style-type: none"> • Analyst • Designer • Architect 	<ul style="list-style-type: none"> • Design Patterns • Software Requirements • QFD 	<ul style="list-style-type: none"> • Extended UML Diagram • Notation for Patterns

Figure 3 Phases in DDP Process

4. Phases in the DDP in more Detail

We describe each of the phases in Figures 1 and 2 in more detail. It is not possible or even desirable to discuss all aspects of the phases here but we do focus on the following issues:

- The activities and steps in each phase
- Traceability in a phase and between phases
- Describing and documenting intermediate and final products in each phase

The idea is simple: each phase corresponds to a core process that accepts input and produces output. The format and structure of the input and output information is known. Furthermore, the steps and activities that are executed are described and documented in a form that is consistent with the language and vocabulary that the involved stakeholders use and can understand.

The four phases can be viewed as four core processes in the software lifecycle. Each process is realised by a system. As we shall see in section 6 each system can be viewed as an instance of a so-called domain category, in this case a Manufacturing (or sometimes Tracking) category. Applications in this category are characterised by a number of defining features the most important of which being that raw materials and raw input are transformed by a series of activities to finished products. Thus, if we view each phase as a system in itself we can describe it in the following terms:

- Action 1: System context (the external systems interfacing with current system)
- Action 2: The core process and information flow in the system
- Action 3: The activities and subsystems

These action points describe each phase. Action 1 describes the external systems (actors) that interface with the current system. Action 2 describes what information is processed in the system and how it is processed. In particular, we

wish to know how 'raw' input information is transformed to so-called internal information, half-product information and final products. Finally, Action 3 describes the activities that are needed in order to transform raw information to final product. This is the stage where we adopt and integrate the entities from the vocabulary. For example, in the analysis phase, we create and integrate the entities from UML.

In a sense, the systems described in the following sections can be seen as glorified compilers. If we keep this in mind we can gain a better understanding of what is going on in each phase. This could be called *learning by analogy* (see [Winston84]).

4.1 Business Modelling

This is the first phase of the software lifecycle and is concerned with determining what the enterprise goals are and how the current system fits into the bigger scheme of things. In particular, we are interested in the long-term goals and objectives in this phase.

The Datasim Development Process does not deal with business modelling at this level of abstraction. It starts with the Requirements Determination phase. However, we determine the so-called domain category of the system under discussion and in particular the business environment in which the system is to operate. We shall discuss the different domain categories in section 6. The main categories are:

- Management and Executive Information Systems (MIS)
- Process Control Systems
- Interactive systems and access control systems
- Resource Allocation and Tracking systems
- Manufacturing systems
- 'Mega' categories (e.g. lifecycle models)

These categories promote understanding between customer and IT personnel and we have seen in practice that discovery of appropriate business models improves the performance of downstream activities (for example, object oriented analysis).

4.2 Requirements Determination

Looking at Figure 2 we see that this system interfaces with the CRS (Customer Requirements Specification), Business Modelling systems (that play a source role) and the Analysis system (that plays the role of sink). In particular, the Requirements system processes the following input:

- Customer requirements document (from CRS)
- Core processes and domain type (from Business Modelling)

The input from CRS is transformed to lower-level and accurate requirements while Requirements synchronises with the Business Modelling system to check if the requirements are consistent with the domain models and vice versa. The deliverables from Requirements represent a set of documented requirements and use cases that will be used as input to the Analysis phase.

The main sub-phases in this phase are ([Sommerville97]):

Requirements Elicitation: the process of discovering the most important system requirements based on the CRS and discussions with key stakeholders. Domain and organisational issues play an important role in this phase. Attention points during elicitation are:

- Consult key stakeholders
- Carry out a feasibility study
- Use goals and core processes to drive elicitation
- Use viewpoints to collect requirements

Requirements Analysis: The initial set of requirements from the Elicitation phase are analysed to spot conflicts, inconsistencies, omissions and overlaps. Once these have been found the requirements analyst must negotiate with stakeholders to resolve these conflicts.

Requirements Documentation: A requirements document is created and distributed to various interested stakeholder groups. We recommend a standard writing style and document layout. The DDP suggests that we document each requirement as follows:

- Requirement Name and ID
- Short description
- Detailed description

- Rationale (reason for requirement)
- Source
- Customer importance (priority)
- Risk factor
- Related viewpoints
- Quantitative description of requirement
- Possible sub-requirements

In fact, we have documented each requirement entity in a standard manner based on the useful guidelines in [Sommerville97]. For more information, see [Datasim2000].

Requirements Management: this phase is concerned with all of the processes involved in changing system requirements. Important attention areas are:

- Traceability policies
- Change management policies
- Recording rejected requirements

Requirements management is one of the least understood and least supported phases in the software lifecycle.

The requirements determination process is depicted in figure 4 and we summarise its main sub-processes:

- (1) The Customer Requirements Specification (CRS) is used to determine what the goals and objectives are in the systems.
- (2) Each goal is realised by one or more core processes. A core process is a sequence of activities that produces products or services that are visible to customers (see [Rummler95]).
- (3) Each core process will be realised by a system. The sub processes in the core process will be realised by the system's subsystems.
- (4) The Customer Requirements Specification (CRS) is used to determine what the main viewpoints in the systems are (see [Sommerville97]).
- (5) We determine which groups of stakeholders and actors are responsible for which viewpoints.
- (6) Each group of stakeholders has its own set of requirements. These requirements are discovered by the so-called Inquiry-based approach, for example.

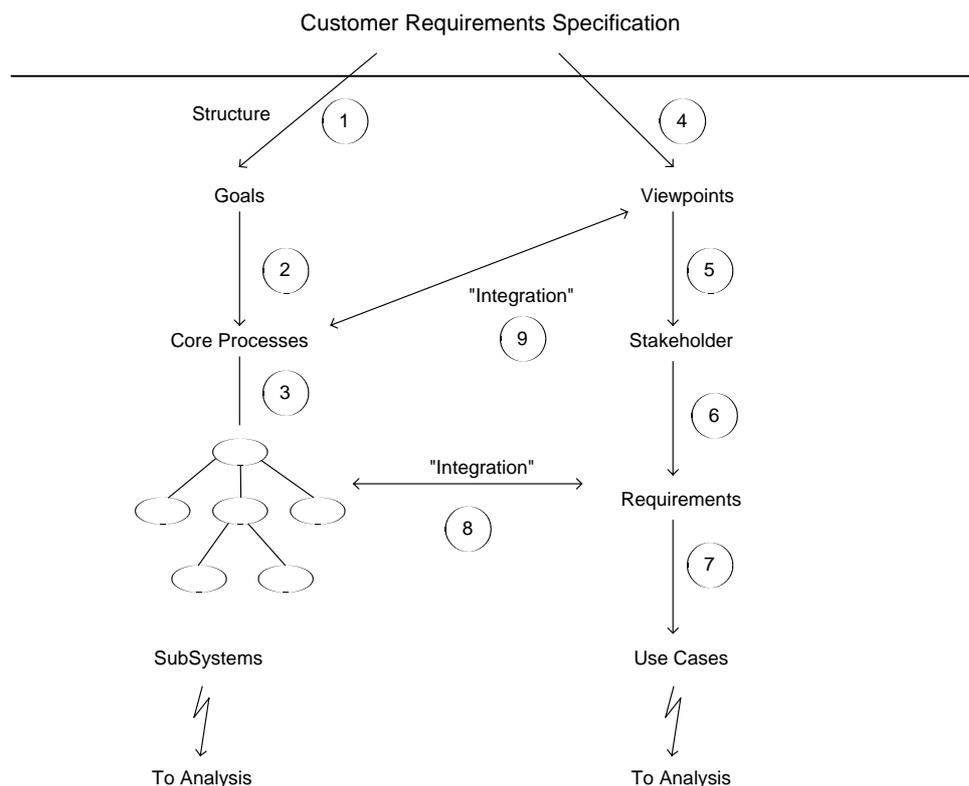


Figure 4 Requirements Determination Phase

(7) A given requirement is realised by one or more use cases. A use case is not a requirement because it describes how the system works in a specific context while a requirement is universal. This part of the process shows major differences with other processes, for example the Rational Unified Process (RUP).

There are two optional extra steps in the process. They are concerned with harmonising system structure with functionality. The basic idea is to ensure that related entities are grouped together:

(8) Associate each viewpoint with a specific process if possible. Ideally, the relationship should be 1:1.

(9) Associate each requirement with one subsystem. This should be carried out in our opinion because different subsystems and their associated requirements can be assigned to independent development teams. The relationship requirement::subsystem is N::1 in general.

We note that the above steps are integrated with the major phases in requirements determination, namely Elicitation, Analysis, Documentation and Management (see [Sommerville97]).

Figure 5 shows the relationships between the different entities in the Requirements phase.

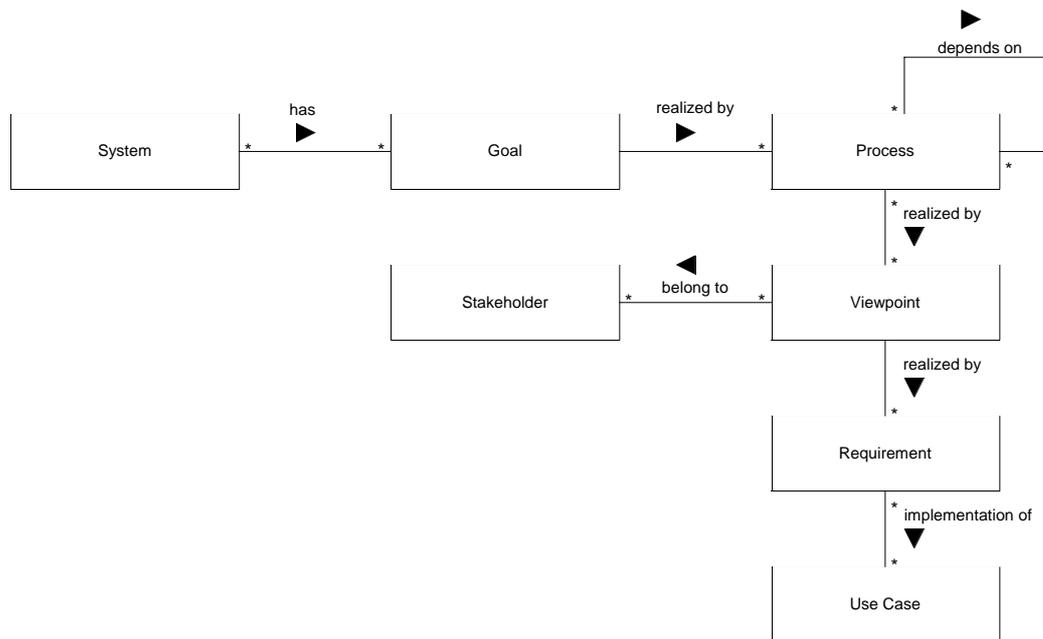


Figure 5: Traceability during Business Modelling and Requirements Determination

4.3 Analysis

This system interfaces with the Requirements and Business Modelling systems (that play a source role) and the Design system (that plays the role of sink). In particular, Analysis processes the following input:

- Use cases (from Requirements)
- Core processes and domain type (from Business Modelling)

The deliverables from Analysis represent a set of diagrams (documented in UML) that will be used as input to the Design phase. We draw a distinction between whether we are analysing using objects or components. The latter approach subsumes the former (components are more general than objects) but the activities that are executed during Analysis are similar. We describe the specific differences in sections 4.3.1 and 4.3.2.

4.3.1 Object-Oriented Analysis

This process is well established within DDP as Datasim has been involved in training and software development since 1990. Since OOA can be viewed in the context of workflow management we can describe the process as a flow of information as described in figure 6. The input to OOA consists of a system decomposition (system and high-level subsystems) and a set of use cases (of course, each use case is a realisation of one or more requirements).

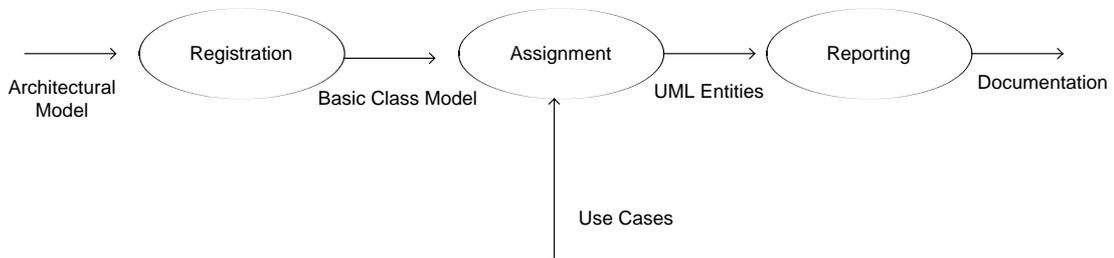


Figure 6 Information flow in OOA

The deliverables or products consist of the following information:

- A full description of each class, including its operations, static attributes and dynamic attributes (possibly represented by statecharts)
- The static relationships between classes in the architecture (associations, aggregations, composition)
- Possible generalisations (inheritance hierarchies)

The process that realises products from input can be seen in figure 7 and is summarised as follows (the numbers are used in order to describe the subprocesses):

- (1) The high-level system architecture is mapped to an initial PAC model containing Boundary, Entity and Control objects (in the sense of [Bass98], [Jacobson99])
- (2) Each use case is mapped to a sequence diagram where the interacting objects arise from the model in (1)
- (3) A precise UML model is created based on the PAC model and based on the validation efforts during the sequence diagram creation phase

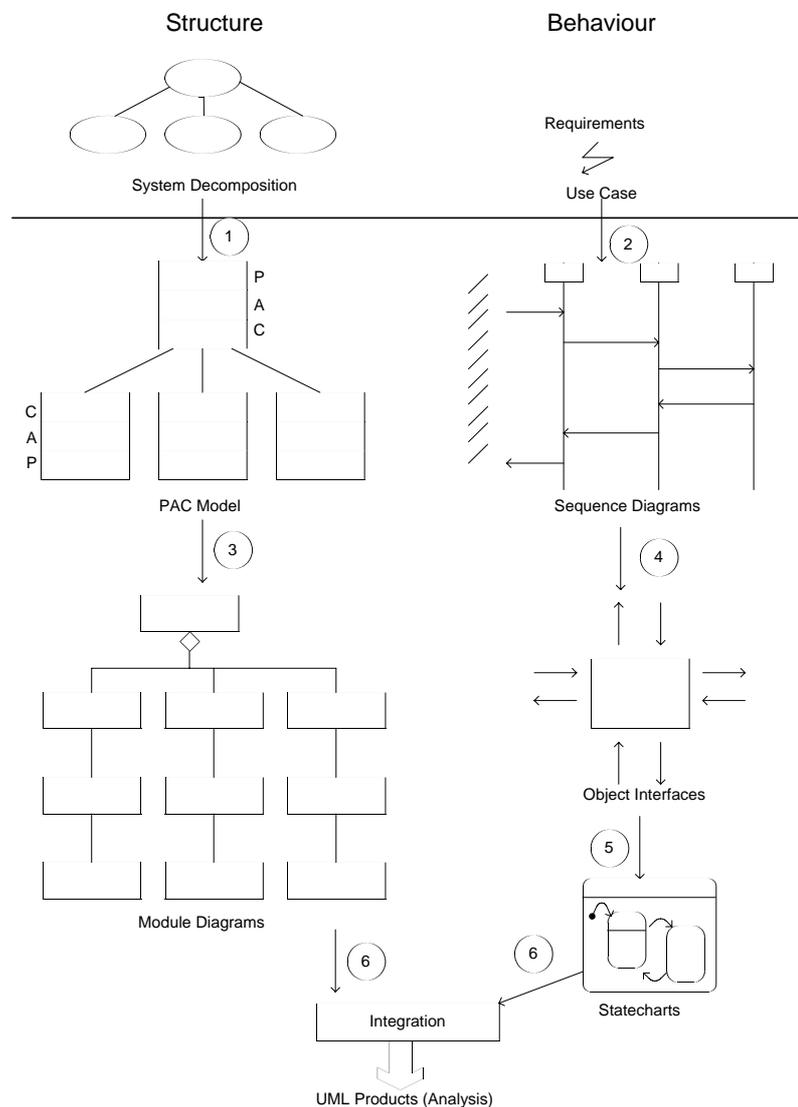


Figure 7 OOA Process (UML)

- (4) Determine what the input and output messages are for each object in the UML class diagrams. An optimisation step is to standardise the message names and/or group semantically-related messages into interfaces or protocols (as in UML for real-time extensions)
- (5) (optional step) Create statecharts for each object in the model.
- (6) Integrate the sub-processes. This sub-process creates the products of OOA.

Datasim has developed a process for real-time applications based on RT UML and it is similar to the above steps (1)-(6). However, there are several major differences because we speak of capsules rather than objects. Capsules have incoming and outgoing protocols that are defined by so-called ports. A port is a realisation of a protocol. A protocol is an abstract specification of a group of semantically related messages.

4.3.2 Component Oriented Analysis ®

An advantage of our process is that once the OOA process has been learned it is fairly straightforward to adapt it to cater for components. Object technology and component technology are similar, yet different. The major differences between the two technologies are:

- Object technology is finely grained in the sense that the central entity to be discovered during OOA is the object. An object has operations and attributes. Component technology, on the other hand concentrates on discovering the contracts and interfaces between subsystems. A component is an entity that realises one or more interfaces and it can be high-level or low-level. Objects tend to be low-level in general.
- Object oriented analysis has evolved as a bottom-up process. If you want proof, just talk to those who have learned OO the hard way. A good example can be found in IEEE Software 1999 where the initial design of the Mars Pathfinder software is described. Component Oriented Analysis, on the other hand, can be developed in a top-down or bottom-up fashion, thus leading to maintainable and flexible systems.
- Object technology is a precondition for successful component development. It is not possible in our opinion to develop components without first having a good understanding of objects.

The main steps in COA are similar to those as discussed for OOA as shown in figure 7. The major difference is that more emphasis is placed on inter-component communication (interfaces) and less emphasis on the entities (whether they be objects or components) that realise those interfaces.

5. Design

Whereas the Requirements Determination and Analysis phases are concerned with describing the problem domain the Design phase is concerned with creating an architectural framework that accommodates the software and hardware in the system. In other words, we wish to know how to describe the eventual implementation by constructing blueprints that will be mapped to implementation languages such as Java, C++ and C#. To this end, we discuss three important issues:

- The scope of the Design phase
- The information flow in the Design phase
- Creating standardised design 'blueprints'

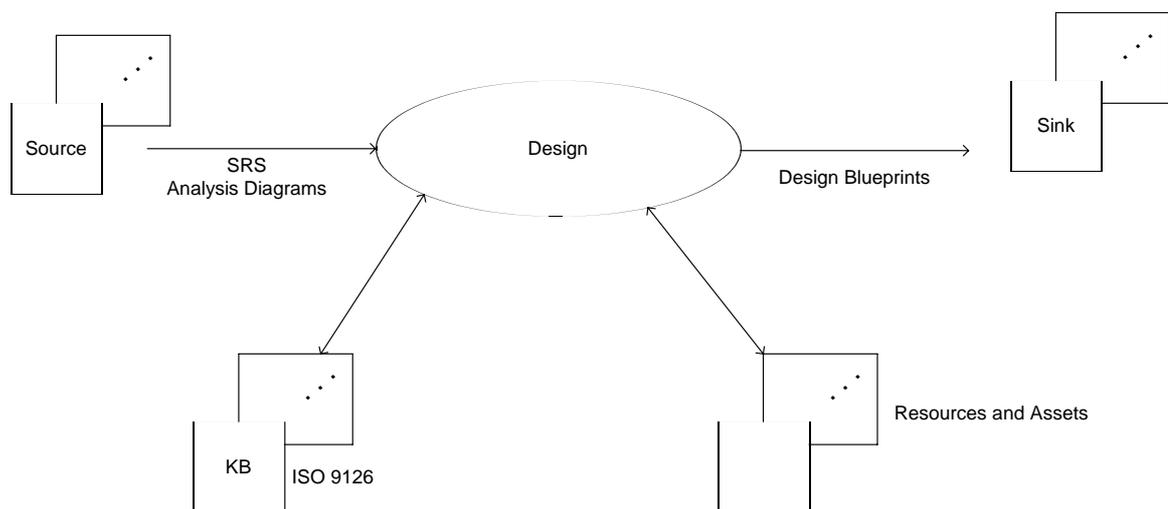


Figure 8 Context Diagram for Design Process

In order to describe the scope we resort to a context diagram as shown in Figure 8. This diagram describes the relationship between Design and its clients and servers. These are:

- Source: the systems that deliver input to Design; first, a Software Requirements Specification (SRS) that describes the software and hardware requirements that the system should satisfy. The SRS is considered to be primary input and Design becomes an instance of a Resource Allocation and Tracking (RAT) category because high-level and abstract requirements are transformed into more concrete requirements and eventually design blueprints. The second input (secondary input) consists of the output from Analysis, namely class diagrams, sequence diagrams and Statecharts (see [Jacobson99])
- Sink: systems that are the recipients of the products from Design. Typical examples are C++ systems that map the design blueprints (for example in the form of UML documented design patterns) to executable code. Other Sink systems are also possible.
- Knowledge Base (KB): classification systems that contain information about generic requirements and past software projects. For example, we use the ISO 9126 quality characteristics in order to measure software quality. The incoming software requirements in the SRS are validated against the ISO 9126 characteristics; some requirements may be eliminated, others will be generalised while yet others may be merged.
- Resources and Assets: the systems that contain knowledge of how to realise high-level requirements, which software developers are available and so on. This system could be used as the basis for scheduling and outsourcing activities.

We have integrated both the Gamma (GOF) and the Buschmann patterns into DDP (see [Gamma94], [Buschmann96]) and the patterns in those books form the basic ingredients for our design blueprints.

Having described the context diagram we must now describe how high-level requirements are mapped to design blueprints. There are three main subprocesses, namely Registration, Assignment and Monitoring as shown in Figure 9. Registration is responsible for the validation of the requirements in the SRS and mapping them to internal requirements (for example, the ISO 9126 characteristics). The Assignment process is responsible for the alignment of the internal requirements with the class and/or component models from the Analysis phase. In particular, we adopt the Design Dimension approach (see [Shaw96]) as a useful and convenient means for discovering requirements. Furthermore, a given design dimension is realised by several design and architectural patterns. Finally, the Monitoring process 'displays' the half-products (in this case design patterns) in different forms, for example as UML diagrams, source code stubs and so on.

To summarise, we track high-level software requirements by mapping them to lower-level requirements. We use four levels and it is our feeling that this number is sufficient. They are:

- Software requirements from SRS
- ISO 9126 characteristics (see [Kitchenham96] and the Appendix below)
- Design Dimensions (for example, [Shaw96])
- Design blueprints (for example, the GOF, Buschmann and Schmidt patterns)

We thus have three mapping rules and these can be documented by the QFD method (see [Cohen95]). The mapping rules can also be quantified.

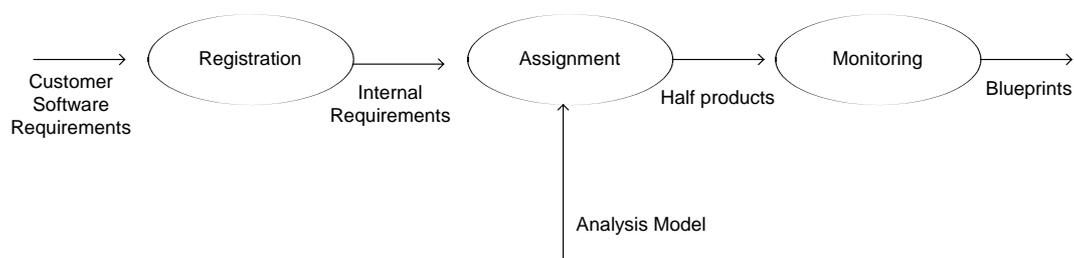


Figure 9 Information Processing in Design

6. Domain Categories

The present author has invented and documented a number of so-called design categories. A design category 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 have 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 (INT)

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 (as was promised in the early hype days of OO) but in the analysis and design phases.

6.1 Information Engineering

We assume that a core process is mapped to a single system. Our goal is to discover how input information is transformed into final output in a series of steps. Each step is realised by a subsystem. In essence, we are modelling the information in a system and this area is called Information Engineering. A definition can be given as follows (see [Martin92]):

'Information Engineering (IE) is the application of an interlocking set of formal techniques for the planning, analysis, design and construction of information systems in a whole enterprise or across a major section of the enterprise'

From this definition we conclude that it should be possible to model large-sized, medium-sized and even small systems using IE techniques. IE is consistent with the top-down approach taken in previous sections because with IE high-level plans and models are created and separately built systems are linked into these plans and models. IE progresses into a top-down fashion by executing the following steps (see [Martin92]):

- Modelling the enterprise
- More detailed modelling of business areas
- Individual business planning
- System design
- Construction

Modelling the enterprise corresponds to the area called Information Strategy Planning and is concerned with top management's business goals and critical success factors in combination with a high level overview of the enterprise with the associated functions, data and information needs. Business modelling is realised by business area analysis that determines what processes are needed to run a given business. In particular, we must determine how the processes are interrelated and what data is needed. System design is concerned with the problem of determining how processes and data from the business area will be implemented in specific procedures or services. Finally, construction involves the implementation of procedure using code generators, for example.

The relationship between IE and this white paper is that this paper is concerned with the information flow in a given business area. We could say that we are modelling information at the intermediate level in the sense that it is neither the enterprise level nor system-level that we are interested in. Our starting point is to map core processes to other forms. In particular, for each process we must determine:

- The input information to the process
- The output information from the process
- Create a system that is responsible for processing input and delivering output

We define a system as a computational entity that is capable of processing input information from several sources and producing output information that will be used by consumers in a value chain. We distinguish between two categories of input information: first, primary information corresponds to the information that is needed by the core process and by definition is indispensable. On the other hand, secondary information represents input that is used in conjunction with the primary input in order to produce the necessary output information. When developing business system models it is sufficient to concentrate on the primary input information while secondary input will be discovered in later stages of the software lifecycle when we discuss requirements and use cases. Primary input allows us to discover the high-level business architecture and the 'architecturally important' use cases (see [Jacobson99]).

The attention points at this stage of development are:

- Discover the names and high-level formats of all inputs and outputs
- Determine the 'supplier' system that delivers primary input (producer)
- Determine the 'customer' system that receives output (consumer)
- Determine the name of the system that produces output from input

6.2 Some Important Applications

We give a summary of some subclasses and instances of the domain categories. The basic classification system is still evolving (usually in synch with the author's mental development as an analyst). 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 (INT)

- 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.

7. Conclusions

We conclude this article by summarising what we think is the added value of the DDP. Instead of listing a number of specific advantages we group them into the following categories:

- Software Lifecycle support
- Defined process for UML (in analysis phase)
- Integration with existing paradigms (e.g. Design Patterns)
- Improved communication and less project fiascos

Our eventual aim is to create a seamless and transparent mapping from products in each phase of the software lifecycle to other products in succeeding phases. The mapping rules should be made explicit and they can be used by less experienced developers in much the same way that mechanical engineers use handbooks and drawings to design new motors, compressors and other machines. We are already beginning to see the emergence of 'software handbooks' (see [Gamma94], [Buschmann96]) but more work needs to be done in the area of institutionalisation of reusable knowledge and expertise in all phases of the software lifecycle. In this way we hope that the IT can catch up on other disciplines where standard procedures are in place to help professionals develop products that are robust, reliable and satisfy customer requirements.

My views, criticisms and suggestions concerning OO are summarised as follows:

- More domain analysis support
- More support for architecture
- Less syntax
- Ban multiple inheritance (it doesn't work)
- Fewer objects and more components
- More discussion of success stories

- More discussion of failure stories (at least we know what not to do!)
- Rehabilitate Information Hiding and system decomposition into OO
- OOT is not necessary and not sufficient for success in enterprise development (in the way mathematicians use these terms)
- Use cases need major revision

I look forward to comments and feedback.

Appendix: The ISO 9126 Characteristics

The ISO 9126 standard has six measures of software quality. They are:

- *Functionality*: does the product satisfy stated or implied needs?
- *Reliability*: does the product maintain its performance level under stated conditions for a stated period of time?
- *Usability*: the effort needed to use the product by a stated or implied set of customers?
- *Efficiency*: how does the product perform when compared with the resources needed to carry out the stated tasks?
- *Maintainability*: the effort needed to make specified modifications (including corrections, improvements or adaptations in the operating environment). This characteristic includes the problem of adapting to changes in requirements.
- *Portability*: the ability to transfer software from one environment to another one. This includes organisational, hardware and software changes.

About the Author

Daniel Duffy (1952) is the founder of Datasim Education BV and Datasim Component Technology BV, Amsterdam, the Netherlands. These companies are involved in training, consultancy and software development using object-oriented, component and agent technologies. He has been involved in the IT branch since 1979 and has worked in the semiconductor, engineering, oil and gas, financial and process industries as programmer, systems analyst, network manager and designer. His current interests are in the industrialisation and quantification of the software process by discovering and documenting successful projects from the past and using them as reusable frameworks for future systems. At the moment of writing he is involved with the development of mathematical models for financial derivatives pricing and designing these models using patterns and C++.

Daniel Duffy has a M.Sc. (1977) and Ph.D. (1980) in numerical analysis from Trinity College, Dublin, Ireland. He spent some time as a researcher in various European countries before he switched to IT.

Daniel J. Duffy, Ph.D.

Datasim Education BV

ddduffy@datasim-education.com

Notice

Datasim Development Process, Component Oriented Analysis, COA, Component Oriented Design and COD are registered trademarks of Datasim Education BV, Amsterdam, the Netherlands.

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

