

# *2 Domain-Orientation in Software Development*

---

During the difficult process of developing a computer application, a real-world oriented view and a computer oriented view must be brought together in order to end up with a useful software product. The real-world oriented view focuses on the part of the real world where the application is to be used. This view on subject-matter concerns is necessary to be able to build a product that satisfies the needs of the user. In contrast, the computer oriented view focuses on structures and properties of programming constructs. This view is indispensable because a program must be written that can run on a computer.

*Domain-oriented* software development emphasizes the real world view. Software solutions are based on rigorous system descriptions that capture properties and structures of the application domain. The computer oriented view becomes important in a subsequent development phase when the domain-oriented solution is implemented on a computer by means of a programming language. The benefits of domain-oriented development are the comprehensibility of solutions, a low risk of fundamental design errors, and simplified maintenance.

In contrast, conventionally developed software solutions are mainly determined by the computer oriented view because the final program is the only rigorous system description. Although conventional software development methods support domain-orientation by initial analysis models, these system descriptions have a minor impact on the final solution because they are not rigorous.

### 2.1 The Software Engineering Problem

When we develop a computer application, we are constructing a machine that must bring about certain useful services and functions in its real environment. The machine may be, for example, a simple text editor which allows its user to create and manipulate texts that can be sent to a printer. A more complex example of such machine is an embedded system that controls and coordinates a number of mechanical machines.

Figure 1 illustrates the problem to be solved. The graphical notation stems from Michael Jackson's approach to problem analysis (Problem Frames). We shall use this approach in chapter 3 when we analyse the embedded system problem.

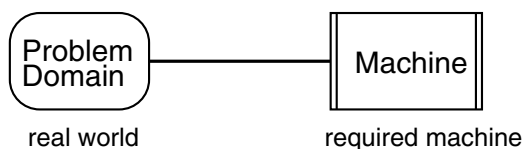


Figure 1: The given problem domain and the machine to be built

The domain connected to the machine is the part of the world in which the machine's computations have a useful meaning or effect; it contains everything that will interact with the machine or furnish the subject matter of those computations. This domain is the *problem domain*. In the text editor example, the problem domain comprises the user and the edited texts; in the embedded system example, the problem domain consists of the mechanical machines to be controlled, of peripheral devices, and probably of an operator which can send control commands to the embedded system and receive its feedback.

The problem we must solve is located in the world outside the machine we build. The machine must bring about certain requirements that concern things and phenomena of the problem domain. The problem domain is the given part of the software engineering problem while the solution task is to build the required machine.

Figure 2 shows how the required machine is fabricated. Our main resource is a computer. By writing a computer program, we are able to specialise this general purpose machine so that it realises the required machine.

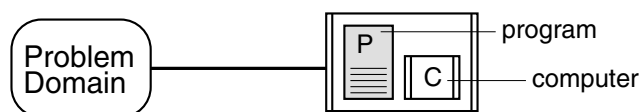


Figure 2: The required machine implemented on a computer

The program is a description that expresses two things. First, it describes explicitly the implementation of our solution; that is, it specifies how the computer must execute. Second, it describes implicitly how the built machine behaves in the real world; that is, it determines how inputs and outputs are operationally related to bring about the required behaviour in the problem domain. Thus, programming must always be guided by at least a mental model of the required machine. Even for small problems, it is not possible to think of the required machine in terms of program statements only.

### *When programming gets difficult*

Programming gets difficult when we try to develop our problem solution directly by programming. A main reason is that the program statements are not directly related to the particular problem domain. In order to understand what effects our program has to the real world, we must deduce from the program text how the outputs are caused by the inputs, and we must know how the outputs and inputs are related to the problem domain. In order to be more efficient in program understanding, we try to establish direct relationships between the program text and the problem domain. For simple problems, such relationships can be described in the form of program comments, supported by well-chosen names of variables, procedures and objects.

However, for serious software development, the ‘method’ of basing domain-orientation on a programming style definitely fails. Many difficulties in software development are related to the fact, that we do not clearly distinguish between domain- and implementation-oriented aspects of the solution. How a solution can be described depends on the problem to be solved. A domain-oriented solution description of a software engineering problem is typically a structure of operational rules that can be implemented in any programming language. The meaning of the rules must be related to the problem domain in order to allow verifying that the machine satisfies the requirements. For example, the solution of the mentioned text editor problem can be based on a text model that is updated in correspondence to user inputs; or the solution of a simple control problem can be described by a finite state machine. Such operational solution descriptions are often called operational specifications.

A fundamental difficulty of programming is that the domain-oriented structure of the solution often does not fit to the structure of the program. A domain-oriented structure reflects real structures of the problem domain whereas the program must be based on programming language constructs. What we need is an explicit domain-oriented description of our solution. Writing the program is still a remaining problem, namely the problem of implementing the domain-oriented solution.

## 2 Domain-Orientation in Software Development

---

In addition to the difficulty of developing and understanding a program, there is the serious challenge in maintenance. Software maintenance mainly deals with changes of functional requirements. The meaning of new functions is defined in the problem domain, but the necessary changes must be made in the program text. Thus maintenance becomes difficult when the necessary relationships to the problem domain are described insufficiently, and, above all, when the structure of the new functions clashes with the structure of the existing program. The high costs and risks encountered in software maintenance are mentioned in most papers and books on software engineering.

### *2.2 Domain-Oriented Software Development*

As discussed above, by formulating the solution directly in a programming language, we encounter the difficulty of expressing clear relationships to the problem domain. This difficulty arises because the semantics of a programming language concerns only the internal behaviour of the computer.

Another more central challenge in software development is to find the ‘good’ solution. There are infinitely many programs possible that implement the required machine correctly. However, a program that just runs correctly is not necessarily a good solution. The central questions are “what is a good solution?”, and “how can we find a good solution?” Domain-orientation gives an answer to these questions.

#### *2.2.1 The Domain-Oriented Development Principles*

The essence of the paradigm of domain-orientation is expressed by means of two development principles.

##### **Description Principle**

Solutions are to be described by rigorous descriptions that relate to the problem domain.

##### **Correspondence Principle**

The structure of the solution is to be based on properties of the problem domain.

The *description principle* stipulates that software development has to be based on domain-oriented system descriptions that are rigorous. Such system descriptions define solutions in terms that are related to the problem-domain. Domain-oriented descriptions techniques – though they are not rigorous – are

well known from analysis models of conventional software development methods; examples are data flow diagrams, entity-relationship diagrams and state transition structures (see 2.2.4). To implement a domain-oriented solution on a computer we additionally need an implementation-oriented system description, the program. For certain problem classes it may be possible that program can be used to express both domain- and implementation-oriented concerns.

The implementation of the solution defines how the computer must execute; hence, this description must be rigorous as a matter of fact. But the domain-oriented description must be rigorous as well. Both descriptions describe the same subject, the solution. We therefore must guarantee that the two descriptions remain consistent during the development process. If the domain-oriented description is not rigorous – which is the case in the mentioned conventional analysis models – we cannot verify the consistency of the two descriptions properly. A domain-oriented description that is not rigorous must therefore be expected to become obsolete in early iterations of the software development process already, and as a result, the final solution is defined by the implementation only.

The *correspondence principle* addresses the main goal of domain-orientation: the structure of the solution is to be based on properties of the problem domain. The nature of the treated problem crucially affects which properties of the problem domain are relevant. For example, a railway cross control system should be based on the topological structure of the physical rail system. For a flight reservation system, the relationships between flights, planes, seats and passengers play a key role. For an embedded system, it is the physical behaviour of the controlled external processes that is relevant for the development of the control functions.

The main benefit of explicit correspondences between problem domain and solution space is comprehensibility. We understand a solution when we can see how it brings about the requirements. Requirements address properties of the problem domain; thus to understand a solution we must relate the solution description and the problem domain. Making properties of the problem domain explicit in the solution description will obviously facilitate the task of understanding the solution.

An equally important benefit is the gain of structural robustness during the whole software life cycle, especially when it comes to functional system extensions and maintenance activities. Changes of the functionality are much more frequent than changes in the problem domain. Therefore, solutions based on structures of the problem domain are likely to be more robust than solutions

## 2 Domain-Orientation in Software Development

---

based on structures that have been designed to bring about a particular system functionality.

### *Software Engineering versus Classical Engineering Disciplines*

Software engineering differs from classical engineering disciplines mainly because the developed products are intangible. The final product of a software project is a description: the program text. The products resulting from intermediate engineering activities are system descriptions too. Nevertheless, too often we run into the trap of solving engineering problems directly by programming because usually there is no clear cut between software engineering activities and programming. In addition, due to the expressive power of programming languages, software professionals develop programs for nearly every imaginable application domain. Correspondingly, most software development methods have been designed as general purpose methods, typically based on a general purpose programming paradigm. Thus we cannot expect such methods to capture the specific nature of a particular problem type.

In contrast, classical engineering disciplines do not suffer from such difficulties. The main reason is that the developed products are tangible. The conceptual work of describing a solution and the physical fabrication of the product are fundamentally different activities that are performed by correspondingly skilled professionals. A mechanical engineer, for example, describes the machine to be built in terms that abstract from fabrication details while the fabrication of the machine is the task of a team of highly qualified craftsmen. Furthermore, when the existing machine must be changed or extended, it is common practice to involve the mechanical engineer to elaborate the new solution. In addition, classical engineering disciplines profit from important professional specialisation. Every engineering branch uses established description techniques to formulate domain-oriented solutions.

### *2.2.2 Domain-Oriented Description by means of Abstract Machines*

An attractive way to solve the conflict between domain- and implementation-oriented system descriptions is to describe the required machine rigorously as an abstract machine. An abstract machine is a mathematical object that can be executed symbolically. Such system models are also called operational system specifications. Well known examples of abstract machines include state machines, Petri nets, regular action expressions (structograms) and mathematical descriptions of dynamical system. Figure 3 illustrates the relation of the domain-oriented and the implementation-oriented description level.

## 2.2 Domain-Oriented Software Development

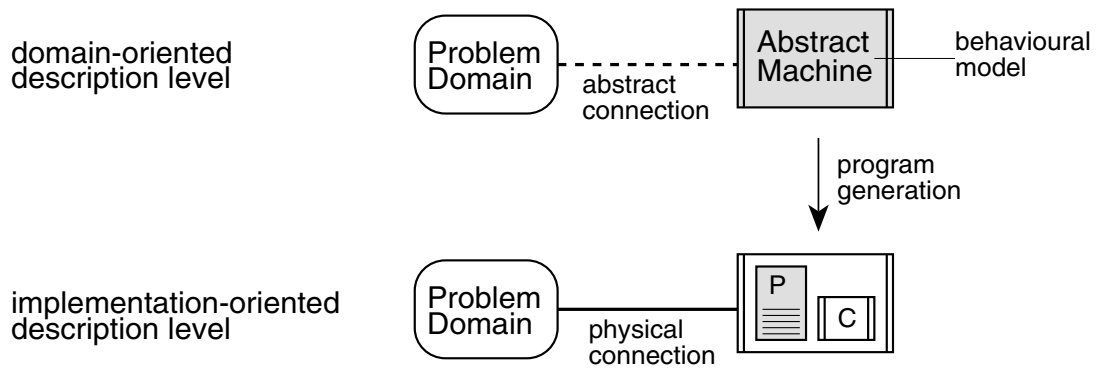


Figure 3: Domain-oriented description of the problem solution by an abstract machine

Abstract machines are suited for domain-oriented system description because we directly construct a model of the required machine. Events and states of a state machine, for example, represent abstract symbols that can be related to corresponding phenomena of the problem domain. Furthermore, we expect that it is possible to build software tools that transform descriptions of abstract machines into executable programs. Due to the tool-based generation of the implementation the consistency to the domain-oriented description can so be maintained automatically.

Unfortunately, for real software development problems it is often not feasible to describe the solution completely by an abstract machine. The formulation of the complete solution usually addresses many different concerns, such as algorithms, reactive behaviour and graphical user interfaces. Abstract machines are in general not suited for the solution description of all arising subproblems. However, an abstract machine can often serve as a domain-oriented executable framework that is used to integrate solutions of particular subproblems. These integrating solutions are described by other suitable description techniques.

For embedded systems, the development concepts formulated in chapter 3 describe how the essential part of an embedded system problem can be solved by an abstract reactive machine, and how this partial solution is completed by integrated computational primitives for data processing and control algorithms.

### 2.2.3 Domain-oriented Methods

The main goal of domain-oriented software development is to base the formulation of solutions on structures that reflect essential properties of the problem domain. Because the problem domain of a complex software system usually addresses many different concerns, the first task of domain-oriented software development must be to separate subject matters and to decompose the prob-

## 2 Domain-Orientation in Software Development

---

lem to be solved correspondingly. The goal of this task is to end up with a structure of tractable subproblems. The decomposition determines a gross software architecture that is stable during the development process. Furthermore, the problem decomposition allows the developer to choose domain-oriented methods that are tailored to the type of the particular subproblems. Domain-oriented methods are able to provide maximal support for the construction of intelligible solutions because they exploit the structure and properties of the addressed problem type.

A domain-oriented method is expected to declare and describe clearly the problem type it is tailored to. In addition, the method must implement the two domain-oriented development principles formulated above. First, the *description principle* is to be supported by a suitable domain-oriented description technique that is rigorous. The purpose of such languages and modelling frameworks is to allow the developer to describe solutions in the clearest and most direct way. Second, the *correspondence principle* is to be specialized by stipulating the kind of properties of the problem domain the solution structure is to be based on. The proposed type of correspondences represents a main development concept of a domain-oriented method.

If the implementation language is suited to describe partial solutions of certain subproblems, a domain-oriented method can also provide rules of how these partial solutions are to be integrated in the domain-oriented description language. In order to support a pragmatic approach, this technique can even be used as a work-around to solve subproblems that are not addressed by the method.

In the following, we give three kind of examples of domain-oriented development techniques used in practice. *Domain-oriented description techniques* support the description principle, *data modelling* techniques support the correspondence principle, and finally the *JSD* is an example of a software development method which supports both principles of domain-orientation.

First, *domain-oriented description techniques* allow the developer to formulate implementation-independent problem solutions. Such system description techniques are used in practice because they are better suited to describe solutions than programming languages. They are in general not accompanied by any methodological development principles. *SDL* [REF], for instance, is a formal specification language that has been used since the 80's to model communication protocols. *Statecharts* [REF] is an example of a visual language used to construct reactive systems by means of hierarchical state machines. *Applicative functional programming* [REF] is a formal description technique that defines computations by means of functional equations.

Second, *data modelling* is an old and widely used domain-oriented development technique that is based on correspondences between problem domain and software system. Commercial information systems, for example, are based on a data model of problem domain entities. Data structures defined in a programming language serve as a domain-oriented description. The maintained structural correspondence is a mapping of real world entities and their attributes to the data model incorporated in the developed program.

Third, *JSD (Jackson System Development)* [REF] is an example of a genuine domain-oriented method supporting both the description and the correspondence principle. The method supports the development of information systems that monitor a part of the real world, and that respond to enquiries about the stored information. Classical examples of such systems are computer-based library systems, payroll systems and accounting systems. A system is modelled by a collection of sequential processes that communicate via datastreams. The communication between processes is specified in a Network Diagram while the behaviour of a process is described by means of *structograms*, a diagrammatic notation of executed action sequences. Solutions are based on behavioural correspondences between the problem domain entities and the processes of the system model. A correspondence is defined by a structogram which describes the set of action sequences that are written by a real world entity and must be read by the connected process.

### *2.2.4 Conventional Support of Domain-Orientation*

Most software development methods stipulate domain-oriented system descriptions at an early stage of the development. The domain-oriented development phase of conventional methods is usually called analysis, and the produced descriptions are termed analysis models. The commonly accepted purpose of analysis models is to describe ‘what’ the system must do in terms of phenomena and properties of the problem domain.

We shall briefly discuss the analysis models of structured and object-oriented software development approaches. Both approaches have emerged from a corresponding programming paradigm. The analysis models of both approaches are not rigorous. They serve as design sketches of the solution that is defined subsequently by programming.

#### *Structured Development*

Structured Development of Real-Time Systems of Ward and Mellors [REF] proposes an essential system model as initial domain-oriented system descrip-

## 2 Domain-Orientation in Software Development

---

tion. An essential model consists of control and data transformations, connected by control and data flows. Control transformations are modelled as finite state machines whereas data transformations are described by pseudo code. The basic structure of an analysis model is developed in a top down approach by functional decomposition. Thus, the basic structure of the designed solution is invented by the developer; it is not based on correspondences to the problem domain. During the subsequent design and programming phases, the essential model is implemented by means of programming constructs such as tasks and modules.

A basic difficulty with the structured approach is that the analysis model can not be kept consistent with the implementation-oriented design descriptions developed in subsequent development phases. The method proposes to advance from analysis to design by transforming the essential model into an implementation model. However, the semantic gap between analysis and design cannot be bridged properly because analysis models are not rigorous system descriptions. In real projects, the only analysis document updated during upcoming development cycles is usually the context-diagram, which describes the system interface.

### *Object-Oriented Development*

Objects are programming constructs that encapsulate state and behaviour. Object-oriented methods propose *seamless development* by using objects as analysis models, as design elements, and as programming constructs. Bertrand Meyer [REF] expresses it like this:

“... This is why object-oriented designers do not spend their time in academic discussions of methods to find the objects: in the physical or abstract reality being modelled, the objects are just there for the picking!”

The same objects are used in domain- and implementation-oriented system descriptions. The mentioned seamless approach starts with an analysis model, termed object model, that describes how entities of the problem domain are related. In the implementation-oriented design and programming phase, the objects of the object model are elaborated to executable programming constructs.

With the initial object model the developer creates a structural correspondence between the problem domain and the solution; object models are in fact data models with encapsulated behaviour. The established structure captures relations among objects of the problem domain. If these relations represent the essential aspect of the treated problem, the approach works well. This is typi-

cally the case for the development of business information systems or software tools.

For the development of embedded systems, the objects of an object model are enhanced with state transition diagrams. The purpose is to create software objects that mirror the behaviour of the external processes. However, we shall see that for embedded systems, this approach cannot work properly. The main reason is that the interaction among external processes and an embedded system is asynchronous and bidirectional. An external process and its corresponding software object can in general not be expected to have identical behavioural structures. We shall come back to this fundamental difficulty of embedded system development when the notion of behavioural correspondence is discussed (section 3.5).

Furthermore, the interaction between objects of an object model is not modelled, but described by means of method invocation mechanisms of the programming language used in the implementation. This is a major drawback in conventional object-oriented embedded system development. The development of the reactive interaction structure of an embedded system is an essential concern of the given problem, and should therefore be described in a domain-oriented way.

