During the difficult process of developing a computer application, problem-and implementation-oriented views must be brought together to end up with a useful software product. The problem-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. The implementation-oriented view focuses on structures and properties of the program to be constructed. This view is indispensable because we must write a program that can run on a computer.

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

Problem-oriented software development emphasizes the problem-oriented view by demanding problem-oriented system descriptions that are rigorous. Subject matter concerns are captured by basing the structure of solutions on properties of the involved part of the real world. The benefit of problem-oriented development is comprehensibility of solutions, low risk of fundamental design errors, and ease and certainty in the difficult phase of maintenance.
2 Problem-Orientation in Software Development

2.1 The Software Engineering Problem

When we develop a computer application we are building a physical machine that brings about certain useful services and functions in its real environment. The machine is realised by specialising the behaviour of a computer. The figure below illustrates the problem to be solved. 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.

![Figure 1: The given problem domain and the machine to be built](image)

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 machine is the solution to be constructed.

The next figure shows how the machine is constructed. 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](image)

The program is a description that expresses two things: it describes explicitly how the computer executes, and it defines implicitly how the realised machine behaves. 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.

---

1 The graphical notation stems from M. Jackson’s Problem Frame approach to problem analysis.
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. If we know how the inputs and outputs of the program are shared with the problem domain, we can deduce the behaviour of the machine from the program text. However, in order to be able to efficiently construct the program, one tries to establish 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 problem-orientation on a programming style definitely breaks down.

An other difficulty is that the structure of the conceptual solution can often not be fitted to the structure of the program. The structure of a conceptual solution typically reflects real structures of the problem domain, whereas the structure of the program is based on programming constructs. Because the program, describes the implementation, it can often not serve as an explicit description of the provided solution. We consider this description conflict as a fundamental difficulty of software development.

In addition to the difficulty of developing and understanding a program, there is the serious problem of maintenance. Software maintenance mainly concerns changes of the 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 insufficiently described, and, above all, when the structure of the new functions clashes with the structure of the existing program. The high costs and risks thus encountered in software maintenance are mentioned in most papers and books on software engineering.

2.2 Problem-Oriented Software Development

By formulating the solution directly in the programming language, we encounter the difficulty of expressing clear relationships to the problem domain. This difficulty arises because the semantics of the programming language concerns only the internal behaviour of the computer. An other more central difficulty of software development is to find “the good solution”. There are infinitely many programs possible that realise the required machine correctly. However, a program that runs correctly is not necessarily a good solution. The central
question is “what is a good solution?”, and “how can we find a good solution?” Problem-orientation gives an answer to these question.

2.2.1 The Problem-Oriented Development Principles

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

**Description Principle**

Solutions are to be described rigorously by problem- and implementation-oriented system descriptions.

**Correspondence Principle**

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

The *description principle* stipulates to base software development on two kinds of system descriptions. Problem-oriented descriptions describe conceptual solutions in terms that are related to the problem-domain. Implementation-oriented descriptions describe how the computer must execute. Problem-oriented descriptions techniques 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). Implementation-oriented descriptions are formulated in a programming language, possibly supplemented by configuration descriptions. It is conceivable that for certain problem classes the same description can be used to express both problem- and implementation-oriented concerns.

Both descriptions are claimed to be rigorous. The implementation-oriented description defines how the computer executes, hence, it must be rigorous as a matter of fact. But the problem-oriented description must be rigorous as well. Because both descriptions describe the same subject, the solution, we must guaranty that the two descriptions remain consistent during the development process. If the problem-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 problem-oriented description, that is not rigorous, must therefore be expected to become obsolete already in early iterations of the software development process, and as a result, the final solution is defined by the implementation-oriented description only.

The *correspondence principle* addresses the main goal of problem-orientation: the structure of the solution is to be based on properties of the problem-domain. It crucially depends on the nature of the treated problem which prob-
2.2 Problem-Oriented Software Development

lem domain properties are relevant. 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 airlines, flights, 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 problem domain properties explicit in the solution description will obviously facilitate the task of understanding the solution.

Equally important is the gain of structural robustness in the complete software life cycle, especially when it comes to functional system extensions and maintenance activities. Changes of the functionality are much more frequent then changes in the problem domain. Therefore, solutions based on problem domain structures are likely more robust than solutions based on structures that have been designed to bring about a particular system functionality.

Software Engineering versus Classical Engineering Disciplines. Paradoxically, the difficulty of establishing clear relationships among program and problem domain is made worse due to the immense flexibility of programming languages. There are infinitely many ways of constructing a program for a required machine and too few criteria for choosing a particular way. Furthermore, due to the expressive power of programming languages, software professionals develop programs for nearly every thinkable 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 of such methods much support to capture the specific nature of a particular problem class.

Classical engineering branches do not suffer from this kind of 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. A machine or a civil engineer, for example, describes the machine or the bridge to be built in terms that abstract from fabrication details. Even when the existing product must be extended or changed, it is common practice to involve an engineer or an architect to elaborate the new solution on a conceptual level. In addition, classical engineering disci-
Problem-Oriented Description by means of Abstract Machines

An attractive way to solve the conflict between problem- 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 dynamical system.

Abstract machines are suited for problem-oriented system description because we construct directly 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-oriented description the consistency with the problem-oriented description can so be maintained automatically.

Unfortunately, for real software development problems it is hardly feasibly to describe the required machine completely by an abstract machine. The formulation of the full solution addresses usually 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 problem-oriented framework that is used to integrate solutions of particular subproblems. These integrated
solutions are described by other suitable problem-oriented description techniques.

For embedded system, the development principles 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 Problem-oriented Methods

The main goal of problem-oriented software development is to base the formulation of solutions on structures that reflect essential properties of the problem domain. The properties to be reflected depend on the particular problem to be solved. Because the problem domain of a complex software usually addresses many different concerns, the first task of problem-oriented software development must be to separate subject matter concerns and to decompose the problem correspondingly. The goal of this task is to end up with a structure of treatable subproblems. On one hand, the decomposition determines a gross software architecture that is stable during the development process. On the other hand, the problem decomposition allows the developer to choose problem-oriented methods that fit to the nature of the particular subproblems. By exploiting the structure of the addressed problem class, problem-oriented methods are able to give maximal support for the construction of intelligible solutions.

A problem-oriented method is expected to clearly declare and describe the addressed problem class. In addition, the method should support the two problem-oriented development principles formulated above. The description principles is to be supported by a suitable formal problem-oriented description technique. The purpose of such languages and modelling frameworks is to allow the developer to describe solutions in the clearest and most direct way. Problem-oriented description techniques are tailored to the addressed problem class; they are not expected to be useful as general purpose tools. The correspondence principle is to be supported by stipulating on which kind of problem domain properties the solution structure is to be based. The proposed structural correspondence represents a main development concept of a problem-oriented method.

In practice, problem-oriented system description is supported by various formal description techniques that allow the developer to formulate implementation-independent problem solutions. SDL for instance is a specification
language that has been used since the eighties to model communication protocols. Statecharts is a visual language, used to construct reactive systems by means of hierarchical state machines. Applicative functional programming is a formal description techniques that defines computations by means of functional equations.

An old problem-oriented recipe that supports both the description and the correspondence principle is data modelling. Commercial information systems for example are based on a data model of problem domain entities. Data structures defined in a programming language serve as problem-oriented description technique. The maintained structural correspondence is a mapping of real word entities and of their attributes to the data model incorporated in the developed program.

**JSD (Jackson System Development)**\(^1\) is an example of a true problem-oriented method. The method supports the development of information systems that monitor a part of the real world, and that answer enquiries about the stored information. Classical examples of such systems are computer-based library systems, payroll systems and accounting system. A system is modelled by a collection of sequential processes that communicate via datastreams. Process behaviour is described by means of Structograms, a diagrammatic notation of regular expressions. The asynchronous communication of processes is specified in a Network Diagram. Solutions are based on behavioural correspondences between the problem domain entities and associated process instances of the system model. The correspondences are defined by a number of sets of valid event sequences that are written by the real world entities and read by the associated process instances.

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

### 2.2.4 Conventional Support of Problem-Orientation

Most software development methods stipulate problem-oriented system descriptions at an early stage of the development. The problem-oriented development phase of conventional methods is usually called analysis, and the pro-

---

\(^1\) Experiences with JSD influenced strongly the development of the CIP method.
duced 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. The serve as design sketches of the solution that is defined subsequently by programming.

**Structured Development.** Structured Analysis proposes an essential system model as problem-oriented system description. 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 transformation are described by pseudo code. The basic structure of an analysis model is developed top down by functional decomposition. Thus, the basic structure of the designed solution is invented by the developer; it 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 problem-oriented analysis model can not be hold consistent with the implementation-oriented design descriptions developed in subsequent development phases. The method proposes to make the transition from analysis to design by transforming the essential model into an implementation model. However, the semantic gap among 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 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 problem- and implementation-oriented system descriptions. The promoted seamless approach starts with an analysis model, termed object model, that describe how entities of the problem domain are
related. In the implementation-oriented design and programming phase, the objects of the analysis 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. The established structure captures attributes of business objects and relations among business objects. If the relations among objects represent the essential aspect of the treated problem, the approach works well. This is typically 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 model the behaviour of the external processes. However, we shall see that for embedded system problems, this approach cannot work properly. The main reason is that the interaction among external processes and 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 embedded system difficulty when the notion of behavioural correspondence is discussed (section 1.5).

In addition, object-oriented analysis models do not support any interaction models. Interaction among objects is described by means of method invocation mechanism of the programming language used in the implementation. This is a major drawback in conventional object-oriented embedded system development. The development if the reactive interaction structure of an embedded system is an essential concern of the posed problem, and should therefore not be described in the implementation language.