

# 1

## Introduction



To gain an understanding of some new area, it is virtually unavoidable to be immersed in the material for a while without exactly understanding where it will lead.

<p><i>Principles of Object-Oriented Software Development</i></p> <ul style="list-style-type: none"><li>• themes and variations – <i>object speak</i></li><li>• abstraction – <i>paradigms of programming</i></li><li>• software development – <i>the OO life-cycle</i></li><li>• object technology – <i>trends</i></li></ul> <p>Additional keywords and phrases: <i>object, data abstraction, analysis, design, implementation, distribution</i></p>	<div>1</div>
--	--------------

1-1

Slide 1-1: Introduction

This first chapter will give a preliminary characterization of object-oriented software development, sketch some of its history and give an outline of the main themes of this book. The dominant theme may be summarized by the phrase that object-orientation provides the software developer with the *right* abstractions for the analysis, design, implementation, and perhaps even the testing of complex software systems. The underlying theme of the book, however, is to indicate the technological requirements that must be satisfied to employ these abstractions

effectively in actual software development. Yet another theme of the book is based on the observation that what OO offers is not altogether new. So, we will relate the solutions offered by OO to their precedents in the history of computer programming and software design. The reader may then establish whether OO is just another toy for software developers or a significant contribution to both software engineering and programming.

## 1.1 Themes and variations

Nowadays, many have at least some notion of object orientation. Undergraduate courses teaching programming in Java are becoming standard practice. And, in industry and business, object-oriented technology is being adopted on an increasingly large scale. Nevertheless, to some extent, object orientation is still an emerging technology with many open questions. So, we will start with a brief survey of what object orientation is about, what solutions it offers and what is needed to put these solutions effectively into practice. We will also briefly review some *object terminology*, reflect on the notion of *object computation*, and discuss *design by contract*.

### Themes and variations

- abstraction – *the object metaphor*
- modeling – *understanding structure and behavior*
- software architecture – *mastering complexity*
- frameworks – *patterns for problem solving*
- components – *scalable software*

1-2

Slide 1-2: Themes and variations

**Object metaphor** In an object-oriented approach, objects are our primary abstraction device. Objects provide a metaphor that helps us in each phase of the software life-cycle. During analysis, we may partition the domain into objects, that have properties, possibly responsibilities, and relations among each other. In design, objects are our primary unit of decomposition. In our design, objects may reflect real life entities, such as *Employer* and *Employee*, but may also represent system artefacts, such as *stacks* or *graphics*. In actual development, that is in the implementation, objects are our unit of implementation. Each object itself may be regarded as a collection of functions. But it is the collection of functions, and the behavior that they describe, that we take as our unit; not the individual function.

**Modeling** Taking objects as the unit of analysis, design and implementation, allows us to define the structure and behavior of a software system in a natural

way. Nevertheless, although this may at first sight seem to simplify our task, it does actually become more difficult to develop software. Why? Simply, because it takes more effort to find the right kinds of objects! It is difficult to arrive at stable abstractions, to define the corresponding objects, to define the objects' interfaces and to define the appropriate relations between the objects, and to implement them so that everything works. A consequence of adopting an object-oriented approach is that we have to spend more time in describing and understanding the structure and behavior of the system, and to learn the formalisms and tools that enable us to do so.

**Software architecture** Objects not only provide a metaphor. Objects also define a computational platform. Computation in an object-oriented system consists of objects sending messages to one another. This may give rise to very complicated sequences of instructions, in particular when the system is dependent on events from the outside, for example the window or network environment. To master this complexity, we need to think about how objects can be made to fit together. To benefit from an object-oriented approach, we need to design a software architecture that defines and regulates the interactions between objects.

**Frameworks** When does an object-oriented approach pay off? It does pay off when we have arrived at (more or less) stable abstractions for which we have good implementations, that may be reused for a variety of other applications. A framework is a kind of library of reusable objects. However, in contrast with ordinary software libraries, frameworks may at times take over control. The best-known examples of frameworks are in the GUI domain; frameworks in other domains (e.g. the business process domain) are emerging. Using a framework may simplify your life, since a framework provides generic solutions for a particular application domain. But the price you pay is twofold. You have to understand what (patterns of) solutions the framework provides, and you have to comply with the rules of the game imposed by the framework.

**Components** Frameworks consist of components. Simplistically, components correspond to objects in a one-to-one way. However, life is more complicated. Components usually consist of a collection of objects that provide additional functionality that allows components to interact together. A typical example of components are distributed objects, objects that may be accessed over a network. These objects must have, preferably in a non-visible way, all the functionality needed to make a network connection and send data (arguments and results) over a network.

### 1.1.1 Object terminology

Object-orientation originally grew out of research in programming languages. The first object-oriented language was Simula. However, Smalltalk may be held responsible for the initial popularity of the object-oriented approach. The terminology Smalltalk introduced was at the time unfamiliar and, for many, somewhat

hard to grasp. Nowadays, students and IT specialists, have at least heard the object-oriented jargon. Let's briefly look at it. See slide 1-3. Objects provide the means by which to structure a system. In Smalltalk (and most other object-oriented languages) objects are considered to be grouped in classes. A *class* specifies the behavior of the objects that are its instances. Also, classes act as templates from which actual objects may be created. Inheritance is defined for classes only. From the perspective of design, inheritance is primarily meant to promote the reuse of specifications.

**Object terminology**

object speak

- objects – *packet containing data and procedures*
- methods – *deliver service*
- message – *request to execute a method*
- class – *template for creating objects*
- instance – *an object that belongs to a class*
- encapsulation – *information hiding supported by objects*
- inheritance – *mechanism allowing the reuse of class specifications*
- class hierarchy – *tree structure representing inheritance relations*
- polymorphism – *to hide different implementations behind a common interface*

1-3

Slide 1-3: Object terminology

The use of inheritance results in a class hierarchy that, from an operational point of view, determines the dispatching behavior of objects, that is what method will be selected in response to a message. If certain restrictions are met (see sections ??, ?? and ??), the class hierarchy corresponds to a type hierarchy, specifying the subtype relation between classes of objects. Finally, an important feature of object-oriented languages is their support for polymorphism. Polymorphism is often incorrectly identified with inheritance. Polymorphism by inheritance makes it possible to hide different implementations behind a common interface. However, other forms of polymorphism may arise by overloading functions and the use of generic (template) classes or functions. See sections ?? and ??.

**Features and benefits of OOP** Having become acquainted with the terminology of OOP, we will briefly review what are generally considered features and benefits from a pragmatic point of view. This summary is based on [Pok89]. I do expect, however, that the reader will take the necessary caution with respect to these claims. See slide 1-4.

Both *information hiding* and *data abstraction* relieve the task of the programmer using existing code, since these mechanisms mean that the programmer's attention is no longer distracted by irrelevant implementation details. On the other hand, the developer of the code (i.e. objects) may profit from information hiding

as well, since it gives the programmer the freedom to optimize the implementation without interfering with the client code. Sealing off the object's implementation by means of a well-defined message interface moreover offers the opportunity to endow an object with (possibly concurrent) autonomous behavior.

**Features of OOP****information hiding:** state, autonomous behavior**data abstraction:** emphasis on *what* rather than *how***dynamic binding:** binding at runtime, polymorphism**inheritance:** incremental changes (specialization), reusability

1-4

Slide 1-4: Features of OOP

The flexible dispatching behavior of objects that lends objects their polymorphic behavior is due to the dynamic binding of methods to messages. Polymorphic object behavior is effected by using methods, or in C++ jargon *virtual functions*, for which, in contrast to ordinary functions, the binding to an actual function takes place at runtime and not at compile-time. In this way, inheritance provides a flexible mechanism by which to reuse code since a derived class may specialize or override parts of the inherited specification.

**Encapsulation and inheritance** Object-oriented languages offer *encapsulation* and *inheritance* as the major abstraction mechanisms to be used in program development. See slide 1-5. Encapsulation promotes *modularity*, meaning that objects must be regarded as the building blocks of a complex system. Once a proper modularization has been achieved, the implementor of the object may postpone any final decisions concerning the implementation at will. This feature allows for quick prototyping, with the risk that the 'quick and dirty' implementations will never be cleaned up. However, experience with constructing object-oriented libraries and frameworks has shown that the modularization achieved with objects may not be very stable. Another advantage of an object oriented approach, often considered to be the main advantage, is the reuse of code. Inheritance is an invaluable mechanism in this respect, since the code that is reused seldom offers all that is needed. The inheritance mechanism enables the programmer to modify the behavior of a class of objects without requiring access to the source code.

Although an object-oriented approach to program development indeed offers great flexibility, some of the problems it addresses are intrinsically difficult and cannot really be solved by mechanisms alone. For instance, modularization is recognized to be a notoriously difficult problem in the software engineering literature. Hence, since some of the promises of OOP depend upon the stability of the chosen modularization, the real advantage of OOP may be rather short-lived. Moreover, despite the optimistic claims about 'tuning' reused code by means of inheritance, experience shows that often more understanding of the inherited classes is needed than is available in their specification.

OO = encapsulation + inheritance

1-5

**benefits of OOP**

- *modularity* – autonomous entities, cooperation through exchanges of messages
- *deferred commitment* – the internal workings of an object can be redefined without changing other parts of the system
- *reusability* – refining classes through inheritance
- *naturalness* – object-oriented analysis / design, modeling

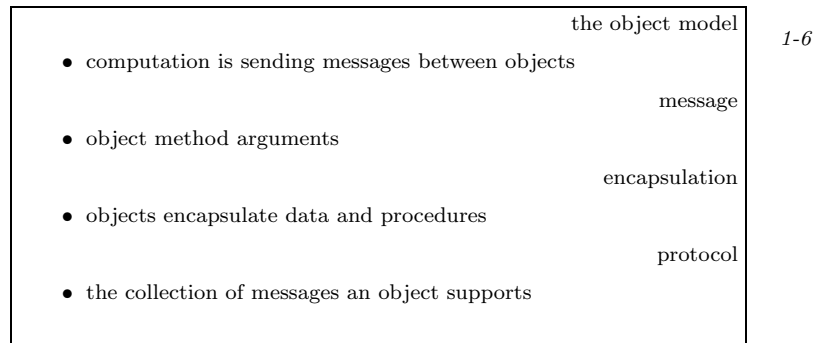
Slide 1-5: Benefits of OOP

The probability of arriving at a stable modularization may increase when shifting focus from programming to design. The mechanisms supported by OOP allow for modeling application oriented concepts in a direct, natural way. But this benefit of OOP will only be gained at the price of increasing the design effort.

### 1.1.2 Object computation

Programming is, put briefly, to provide a computing device with the instructions it needs to do a particular computation. In the words of Dijkstra: *‘Programming is the combination of human reasoning and symbol manipulation skills used to develop symbol manipulators (programs). By supplying a computer to such a symbol manipulator it becomes a concrete one.’* Although we are by now used to quite fashionable computing devices, including graphic interfaces and multimedia peripherals, the abstract meaning of a computing device has not essentially altered since the original conception of the mathematical model that we know as the Turing machine (see below). Despite the fact that our basic mathematical model of a computing device (and hence our notion of computability) has not altered significantly, the development of high level programming languages has meant a drastic change in our conception of programming. Within the tradition of imperative programming, the introduction of objects, and object-oriented programming, may be thought of as the most radical change of all. Indeed, at the time of the introduction of Smalltalk, one spoke of a true revolution in the practice of programming.

The object model introduced by Smalltalk somehow breaks radically with our traditional notion of computation. Instead of regarding a computation as the execution of a sequence of instructions (changing the state of the machine), object-based computation must be viewed as sending messages between objects. Such a notion of computation had already been introduced in the late 1960s in the programming language Simula (see Dahl and Nygaard, 1966). Objects were introduced in Simula to simulate complex real-world events, and to model the interactions between real-world entities. In the (ordinary) sequential machine model, the result of a computation is (represented by) the state of the machine at



Slide 1-6: The object model

the end of the computation. In contrast, computation in the object model is best characterized as cooperation between objects. The end result then consists, so to speak, of the collective state of the objects that participated in the computation. See slide 1-6.

Operationally, an object may be regarded as an abstract machine capable of answering messages. The collection of messages that may be handled by an object is often referred to as the *protocol* obeyed by the object. This notion was introduced in the Smalltalk programming environment originally to provide the means to group the messages to which an object may respond. For instance, the distinction between methods for initialization and methods for modification or processing may be convenient in developing or using a program. The notion of *protocol* may also be given a more formal interpretation, as has been done for instance in the notion of *contracts* (introduced in Eiffel) stating the requirements that must be adhered to in communicating with an object. Structurally, an object may be regarded as a collection of data and procedures. In principle, the data are invisible from the outside and may be manipulated only by invoking the right procedure. In a pure object-oriented language such as Smalltalk and Eiffel, sending a message to an object is the only way of invoking such a procedure. Combined, *data-hiding* and *message interface abstraction* will be referred to as *encapsulation*. Actually, object-oriented languages, while in some way supporting objects as collections of data and procedures, may differ subtly in the degree and way in which they support data-hiding and abstraction.

**Computability and complexity** Mathematically, a computing device consists of a finite table of instructions and a possible infinite memory in which to store intermediate results. In order to perform a computation the device also needs an input and some means by which to display the results. For now, we need not be concerned with the precise mathematical details of our model of a computing device. For a very much more precise and elaborate description of the Turing machine, the interested reader is referred to [Hopcroft]. What is important, however, is that this model captures in a very precise sense the notion of com-

putation, in that it allows us to characterize what can be computed, and also what a computation will cost, in terms of computing time and memory usage. An interesting, but perhaps somewhat distressing, feature of the Turing machine model is that it is the strongest model we have, which means that any other model of computation is at best equivalent to it. Parallel computation models in effect do extend the power of (sequential) Turing machines, but only in a linear relation with the number of processors. In other words, the Turing machine defines what we may regard as *computable* and establishes a measure of the complexity of a computation, in space and time. The awareness of the intrinsic limitations imposed by a precise mathematical notion of computability has, for example, led us to regarding the claims of artificial intelligence with some caution, see [Rabin74]. However, the theoretical insight that a problem may in the worst case not be solved in finite time or space should not hinder us in looking for an optimal, approximate solution that is reachable with bounded resources. An equally important feature of the Turing machine model is that it gives us an illustration of what it means to program a computing device, that is to instruct the machine to perform actions dependent on its input and state. As an extension to the model, we can easily build a *universal* computing device, into which we may feed the description of some particular machine, in order to mimic the computation of that machine. Apparently, this gives us a more powerful machine. However, this has proven not to be the case. Neither does this universal device enlarge the class of computable problems, nor does it affect in any significant sense the computational complexity of what we know to be computable. See slide ??.

#### Computing devices

- mathematical model – *Turing machine*
- universal machine – machines as programs
- computability & complexity – time/space bounded

Object-oriented programming does not enlarge the class of computable problems, nor does it reduce the computational complexity of the problems we can handle.

1-7

Slide 1-7: Computing devices

Interestingly, there is an extension of the (basic and universal) Turing machine model that allows us to extend the narrow boundaries imposed by a mathematical characterization of computability. This extension is known as an *oracle* machine, and as the name suggests, the solution to an (otherwise) intractable problem must come from some external source, be it human, machine-like or divine (which is unlikely). Partly, this explains why *intelligent* systems (such as automatic translation systems) are, to a certain extent, intrinsically interactive, since only the human user can provide the (oracle) information needed to arrive at a solution. Our model of a computing device does quite precisely delimit the domain of com-

putable problems, and gives us an indication of what we can expect the machine to do for us, and what not. Also, it illustrates what means we have available to program such a device, in order to let it act in the way we want. Historically, the Turing machine model may be regarded as a mathematical description of what is called the Von Neumann machine architecture, on which most of our present-day computers are based. The Von Neumann machine consists of a memory and a processor that fetches data from the memory, does some computation and stores the data back in memory. This architecture has been heavily criticized, but no other model has yet taken its place. This criticism has been motivated strongly by its influence on the practice of programming. Traditionally, programs for the Von Neumann architecture are conceived as sequences of instructions that may modify the state of the machine. In opposition to this limited, machine-oriented view of programming a number of proposals have been made that are intended to arrive at a more abstract notion of programming, where the machine is truly at the service of the programmer and not the other way around. One of these proposals to arrive at a more abstract notion of programming is advocated as the *object-oriented approach*. Before studying the intrinsics of the object-oriented approach, however, it may be useful to reflect on what we may expect from it. Do we hope to be able to solve more problems, or to solve known problems better? In other words, what precisely is the contribution of an object-oriented approach? Based on the characterization of a computing device, some answers are quite straightforward. We cannot expect to be able to solve more problems, nor can we expect to reduce the computational complexity of the problems that we can solve. What an object-oriented approach can contribute, however, is simply in providing better means with which to program the machine. Better means, to reduce the chance of (human) errors, better means, also, to manage the complexity of the task of programming (but not to reduce the computational complexity of the problem itself). In other words, by providing abstractions that are less machine oriented and more human oriented, we may enlarge the class of problems that we can tackle in the reality of software engineering. However, we simply cannot expect that an object-oriented approach may in any sense enlarge our notion of what is computable.

**Some history** In the last few decades, we have been able to witness a rapid change in the technology underlying our computer systems. Simultaneously, our ideas of how to program these machines have changed radically as well.

The history of programming languages may be regarded as a progression from low level constructs towards high level abstractions, that enable the programmer to specify programs in a more abstract manner and hence allow problem-related abstractions to be captured more directly in a program. This development towards high level languages was partly motivated by the need to be able to verify that a program adequately implemented a specification (given in terms of a formal description of the requirements of an application). Regarded from this perspective, it is then perhaps more appropriate to speak of a progression of *paradigms of programming*, where a paradigm must be understood as a set of mechanisms and guidelines telling us how to employ these mechanisms. The

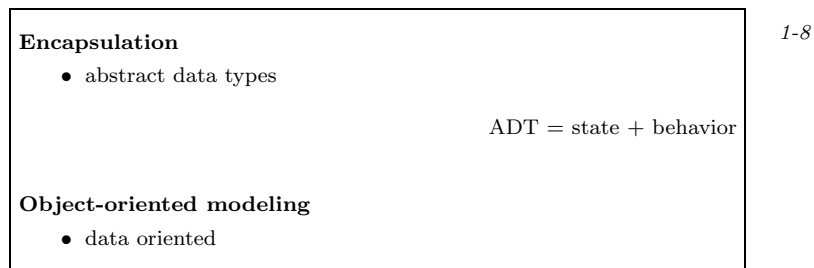
first abstraction mechanism beyond the level of assembler language and macros is provided by *procedures*. Procedures play an important role in the method of *stepwise refinement* introduced by the school of *structured programming*. Stepwise refinement allows the specification of a complex algorithm gradually in more and more detail. Program verification amounts to establishing whether the implementation of an algorithm in a programming language meets its specification given in mathematical or logical terms. Associated with the school of structured programming is a method of verification based on what has become known as *Hoare logic*, which proceeds by introducing *assertions* and establishing that procedures meet particular pre- and post-conditions. Other developments in programming language research are aimed at providing ways in which to capture the mathematical or logical meaning of a program more directly. These developments have resulted in a number of functional programming languages (e.g. ML, Miranda) and logic programming languages, of which Prolog is the best-known. The programming language Lisp may in this respect also be regarded as a functional language. The history of object-oriented programming may be traced back to a concern for *data abstraction*, which was needed to deal with algorithms that involved complex data structures. The notion of *objects*, originally introduced in Simula (Dahl and Nygaard, 1966), has significantly influenced the design of many subsequent languages (e.g. CLU, Modula and Ada). The first well-known *object-oriented language* was Smalltalk, originally developed to program the *Dynabook*, a kind of machine that is now familiar to us as a laptop or notebook computer. In Smalltalk, the data-hiding aspect of objects has been combined with the mechanism of inheritance, allowing the reuse of code defining the behavior of objects. The primary motivation behind Smalltalk's notion of *objects*, as a mechanism to manage the complexity of graphic user interfaces, has now proven its worth, since it has been followed by most of the manufacturers of graphic user interfaces and window systems. Summarizing, from a historical perspective, the introduction of the object-oriented approach may be regarded as a natural extension to previous developments in programming practice, motivated by the need to cope with the complexity of new applications. History doesn't stop here. Later developments, represented by Eiffel, C++ (to a certain extent) and Java, more clearly reflect the concern with abstraction and verification, which intrinsically belongs to the notion of *abstract data types* as supported by these languages.

### 1.1.3 Design by Contract

After this first glance at the terminology and mechanisms employed in object-oriented computation, we will look at what I consider to be the contribution of an object-oriented approach (and the theme of this book) in a more thematic way. The term 'contract' in the title of this section is meant to refer to an approach to design that has become known as *design by contract*, originally introduced in [Meyer88], which is closely related to *responsibility-driven design* (see Wirfs-Brock, 1989). Of course, the reader is encouraged to reflect on alternative interpretations of the phrase *responsibilities in OOP*.

The approach captured by the term *contract* stresses the importance of an abstract characterization of what services an object delivers, in other words what responsibilities an object carries with respect to the system as a whole. Contracts specify in a precise manner the relation between an object and its ‘clients’.

Objects allow one to modularize a system in distinct units, and to hide the implementation details of these units, by packaging data and procedures in a record-like structure and defining a message interface to which users of these units must comply. *Encapsulation* refers to the combination of packaging and hiding. The formal counterpart of encapsulation is to be found in the theory of *abstract data types*. An abstract data type (ADT) specifies the behavior of an entity in an abstract way by means of what are called *operations* and *observations*, which operationally amount to procedures and functions to change or observe the state of the entity. See also section ?? . Abstract data types, that is elements thereof, are generally realized by employing a hidden *state*. The state itself is invisible, but may be accessed and modified by means of the observations and operations specified by the type. See slide ??.

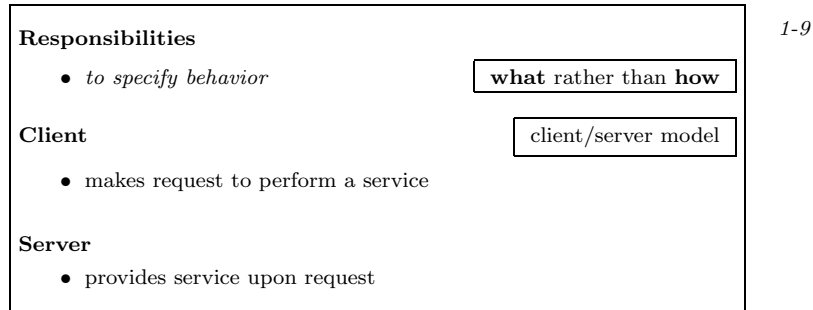


Slide 1-8: Abstract data types – encapsulation

Complex applications involve usually complex data. As observed by [Wirfs89], software developers have reacted to this situation by adopting more data oriented solutions. Methods such as semantic information modeling and object-oriented modeling were developed to accommodate this need. See also sections ??

and ?? . Objects may be regarded as embodying an (element of an) abstract data type. To use an object, the client only needs to know *what* an object does, not (generally speaking) *how* the behavior of the object is implemented. However, for a client to profit from the data hiding facilities offered by objects, the developer of the object must provide an interface that captures the behavior of the object in a sufficiently abstract way. The (implicit) design guideline in this respect must be to regard an object as a *server* that provides high level services on request and to determine what services the application requires of that particular (class of) object(s). See slide ??.

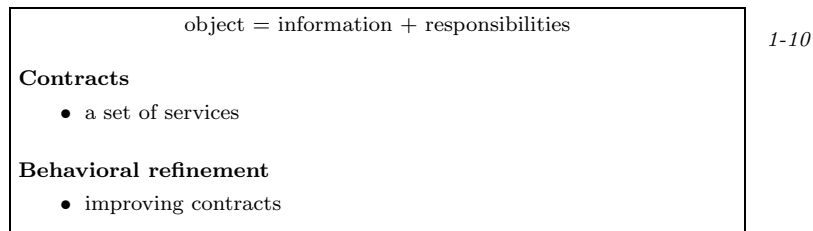
Naturally, the responsibilities of an object cannot be determined by viewing the object in isolation. In actual systems, the functionality required is often dependent on complex interactions between a collection of objects that must cooperate in order to achieve the desired effect. However, before trying to specify these



Slide 1-9: Responsibilities in OOP

interactions, we must indicate more precisely how the communication between a server and a single client proceeds.

From a language implementation perspective, an object is nothing but an advanced data structure, even when we fit it in a client-server model. For design, however, we must shift our perspective to viewing the object as a collection of high level, application-oriented services. Specifying the behavior of an object from this perspective, then, means to define what specific information the object is responsible for and how it maintains the integrity of that information. See slide ??.



Slide 1-10: Contracts and behavioral refinement

The notion of *contracts* was introduced by [Meyer88] to characterize in a precise manner what services an object must provide and what requirements clients of an object must meet in order to request a service (and expect to get a good result). A contract specifies both the requirements imposed on a client and the obligations the server has, provided the requirements are met. When viewed from the position of a client, a contract reveals what the client can count on when the requirements are fulfilled. From the position of the server, on the other hand, when a client does not fulfill the requirements imposed, the server has no obligation whatsoever.

Formally, the requirements imposed on the client and the obligations of the server can be specified by means of pre- and post-conditions surrounding a method. Nevertheless, despite the possibility of formally verifying these conditions, the

designer must specify the right contract for this approach to work at all. A problem of a more technical nature the designer of object-oriented systems faces is how to deal with inheritance.

Inheritance, as a mechanism of code reuse, supports the refinement of the specification of a server. From the perspective of abstract data types, we must require that the derived specification refines the behavior of the original server. We must answer the following two questions here. What restrictions apply, when we try to refine the behavior of a server object? And, ultimately, what does it mean to improve a contract?

**Behavioral refinement** Inheritance provides a very general and powerful mechanism for reusing code. In fact, the inheritance mechanism is more powerful than is desirable from a type-theoretical perspective.

**Conformance – behavioral refinement**

if B refines A then B may be used wherever A is allowed

1-11

Slide 1-11: Behavioral refinement

An abstract data type specifies the behavior of a collection of entities. When we use inheritance to augment the definition of a given type, we either specify new behavior in addition to what was given, or we modify the inherited behavior, or both. The restriction that must be met when modifying behavior is that the objects defined in this way are allowed to be used at all places where objects of the given type were allowed. This restriction is expressed in the so-called *conformance rule* that states that *if B refines A then B may be used wherever A is allowed*. Naturally, when behavior is added, this condition is automatically fulfilled. See slide ??.

The conformance rule gives a very useful heuristic for applying inheritance safely. This form of inheritance is often called ‘strict’ inheritance. However, it is not all that easy to verify that a class derived by inheritance actually refines the behavior specified in a given class. Partly, we can check for syntactic criteria such as the signature (that is, type) of the individual methods, but this is definitely not sufficient. We need a way in which to establish that the behavior (in relation to a possible) client is refined according to the standard introduced above. In other words we need to know how to improve a *contract*.

Recall that from an operational point of view an object may be regarded as containing data attributes storing information and procedures or methods representing services. The question ‘*how to improve a contract?*’ then boils down to two separate questions, namely: (1) ‘*how to improve the information?*’ and (2) ‘*how to improve a service?*’. To provide better *information* is, technically speaking, simply to provide more information, that is more specific information. Type-theoretically, this corresponds to narrowing down the possible elements of the set that represents the (sub) type. To provide a better *service* requires either

relieving the restrictions imposed on the client or improving the result, that is tightening the obligations of the server. Naturally, the *or* must be taken as non-exclusive. See slide ??.

1-12

**attributes**

- more information

**services**

- better services

**contracts**

- more and better services

**A better service**

- fewer restrictions for the client
- more obligations for the server

Slide 1-12: Improving services

To improve a *contract* thus simply means adding more services or improving the services that are already present. As a remark, [Meyer88] inadvertently uses the term *subcontract* for this kind of refinement. However, in my understanding, subcontracting is more a process of delegating parts of a contract to other contractors whereas refinement, in the sense of improving contracts, deals with the contract as a whole, and as such has a more competitive edge.

Summarizing, at a very high level we may think of objects as embodying a *contract*. The contract is specified in the definition of the class of which that object is an instance. Moreover, we may think of inheritance as a mechanism to effect *behavioral refinement*, which ultimately means to improve the contract defining the relation between the object as a server and a potential client.

1-13

**Object-oriented modeling**

- prototyping, specification, refinement, interactions

OOP = Contracts + Refinements

Slide 1-13: Object-oriented modeling

To warrant the phrase *contract*, however, the designer of an object must specify the functionality of an object in a sufficiently abstract, application-oriented way. The (implicit) guideline in this respect is to construct a *model* of the application domain. See slide ??.

The opportunity offered by an object-oriented approach to model concepts of the application domain in a direct way makes an object-oriented style suitable for incremental prototyping (provided that the low-level support is available).

The metaphor of contracts provides valid guidelines for the design of objects. Because of its foundation in the theory of abstract data types, contracts may be specified (and verified) in a formal way, although in practice this is not really likely to occur.

Before closing this section, I wish to mention a somewhat different interpretation of the notion of *contracts* which is proposed by [HHG90]. There contracts are introduced to specify the behavior of collections of cooperating objects. See section ??.

## 1.2 Paradigms of programming

In a landmark paper with the title ‘*What is object-oriented programming?*’ Bjarne Stroustrup raises the question of when a language may be considered to support a particular style of programming, [St88]. See slide ??.

**Object-oriented programming**

- *high tech synonym for good*

**Styles of programming**

- A language *supports* a style of programming if it provides facilities that make it convenient (easy, safe and efficient) to use that style
- compile/runtime checks
- clean interpretation/ orthogonal / efficient / minimal

1-14

Slide 1-14: Styles of programming

In general, one can say that a language supports a particular style of programming if it provides facilities, both syntactic and semantic, that makes it convenient (that is easy, safe and efficient) to use that style. The crucial distinction that must be made in this context is that between allowing a certain style and providing support for that style. Allowing means that it is possible to program in that style. To support a given style, however, requires in addition that suitable compile and runtime checks are provided to enforce a proper use of the relevant language constructs. With these considerations in mind, one could question the assertion that *Ada is object-oriented* or that *Modula supports abstract data types*. Naturally, this attitude backfires with C++. Does C++ support abstract data types and is it really object-oriented?

It is equally important to establish whether a language allows a clean interpretation of the constructs introduced, whether the constructs supporting object orientation are *orthogonal* to (that is independent of) the other constructs of the

<b>Procedural programming</b> <ul style="list-style-type: none"> <li>• procedures, use the optimal algorithms</li> </ul> <b>Modules</b> <ul style="list-style-type: none"> <li>• hide the data, provide functional abstractions</li> </ul> <b>Data abstraction</b> <ul style="list-style-type: none"> <li>• types, provide a sufficiently complete set of operations</li> </ul> <b>Object-oriented</b> – <i>organize your types</i> <ul style="list-style-type: none"> <li>• make commonality explicit</li> </ul>	1-15
---	------

Slide 1-15: Paradigms of programming

language, whether an *efficient* implementation of these constructs is possible, and whether the language is kept *minimal*, that is without superfluous constructs.

Before establishing what the main ingredients of object-orientation are, let us briefly look at some of the styles of programming that may be considered as leading to an object-oriented style. See slide ??.

In his article, Stroustrup (1988) stresses the continuity between the respective styles of programming pictured in slide ??. Each style is captured by a short phrase stating its principal concern, that is guidelines for developing *good* programs.

### 1.2.1 Procedural programming

The procedural style of programming is most closely related to the school of structured programming, of which for instance [Dijkstra76] and [Gries] are important proponents. The procedural style supports a method of program development that is known as *stepwise refinement*. Stepwise refinement is an important heuristic for developing complex algorithms. Instead of writing out a complex algorithm in all its detail, the method allows for refining the elementary steps of the basic algorithm by means of increasingly detailed procedures.

As a playful example of this style of programming, consider the fragment that may be found on the cover of [Knuth92]. See slide ??. Ignoring the contents, clearly the structure shows an algorithm that is conceived as the repeated execution of a number of less complex steps.

### 1.2.2 Data abstraction

When programs became larger and data more complex, the design of correct algorithms was no longer the primary concern. Rather, it became important to provide access to data in a representation independent manner. One of the early proponents of data hiding was, see [Parnas72a] and [Parnas72b], who

```
while ( programming == art ) {  
  
    incr( pleasure );  
    decr( bugs );  
    incr( portability );  
    incr( maintainability );  
    incr( quality );  
    incr( salary );  
  
} // live happily ever after
```

1-16

Slide 1-16: Programming as an art

introduced a precursor to the notion of *data abstraction* as it has become popular in object-oriented languages such as Smalltalk or C++.

As a language that supports data hiding, we may think of Modula-2 that offers strong support for modules and the specification of import and export relations between modules. Also the *package construct* of Ada provides support for data hiding. See slide ??.

Modules as provided by Modula-2 and Ada give a syntactic means for decomposing a program into more or less independent components. It is precisely the purely syntactic nature of modules that may be considered the principal defect of this approach to data hiding. Semantically, modules provide no guideline with respect to how to decompose a program into meaningful components.

**Support for data abstraction**

- Abstract Data Types – *encapsulation*

**Encapsulation**

- initialization
- protection
- coercions

1-17

Slide 1-17: Data abstraction

To express the meaning of a module, we need the stronger notion of *types*, in the sense of *abstract data types* which are characterized by a set of operations. The notion of types as for example supported in CLU, [Liskov74], enables us to determine whether our decomposition satisfies certain formal criteria. For instance, we may ask whether we have defined sufficiently many operations for a given type and whether we have correctly done so. An important advantage of

using abstract data types is that we can often find a mathematical model that formally characterizes the behavior of that type. From the perspective of formal methods, data abstraction by means of abstract data types may be considered as one of the principal means for the specification and verification of complex software systems. See also sections ?? and ??.

From an implementation perspective, to support data abstraction a language must provide constructs to implement *concrete realizations* of abstract data types. Such support requires that means are provided to create and initialize elements of a concrete type in a safe way, and that vulnerable data is effectively protected.

Very important is the possibility of defining generic types, that is types which take a (type) parameter with which they are instantiated. For example, the definition of a *stack* does not differ for a stack of integers, a stack of strings or a stack of elements from an arbitrary user-defined type.

### 1.2.3 Object-oriented programming

There is a close similarity between the object model as presented earlier and the notion of abstract data types just described. Both objects and abstract data types define a set of applicable operations that completely determine the behavior of an object or an element of the data type. To relate an object to an abstract data type we need the notion of *class*, that serves as the description on an abstract level of the behavior of (a collection of) objects. (The objects are called the *instances* of the class.)

As noted in [St88], abstract data types as such, although mathematically satisfying, are rather inflexible and inconvenient for specifying complex software systems. To attain such flexibility, we need to be able to organize our types and express the commonality between them. The notion of class supports this by a mechanism called *inheritance*. When regarding classes as types, inheritance may be seen as introducing polymorphic types. A class that is derived from a particular class (the base class) may be treated by the compiler as a subtype of (the type of) that particular class. See slide ??.

#### Support for OOP

- Polymorphism – *inheritance*

#### Inheritance

- dynamic binding
- protection
- multiple inheritance

1-18

Slide 1-18: Support for OOP

Operationally, the power of inheritance comes from message dispatching. This mechanism is called *dynamic binding*. Message dispatching takes care of selecting

the right method in response to a message or method call. In a hierarchy of (derived) classes, a method for an object may be either defined within the class of the object itself or by one of the classes from which that class is (directly or indirectly) derived. Message dispatching is an essential mechanism for supporting polymorphism, since it allows to choose the most appropriate behavior for an object of a given type. This must occur at runtime, since the type of an object as determined at compile-time may be too general.

An important issue in determining whether a language supports object-oriented programming is whether it offers a protection mechanism to shield the vulnerable parts of a base class from the classes that derived from that class.

Another question of interest is whether a language must support multiple inheritance. Clearly, there is some disagreement on this issue. For example, Smalltalk-80 and Java do not support multiple inheritance. The Eiffel language, on the other hand, supported multiple inheritance from its first days. For C++, multiple inheritance was introduced at a later stage. At first, it was thought to be expensive and not really necessary. Closer analysis, however, revealed that the cost was not excessive. (See Ellis and Stroustrup, 1990.) The issue of multiple inheritance is still not resolved completely. Generally, it is acknowledged to be a powerful and at the same time natural extension of single inheritance. However, the inheritance mechanism itself seems to be under attack. Some doubt remains as to whether inheritance is a suitable composition mechanism when regarded from the perspective of reuse and reliability. An elegant solution is provided by Java which offers multiple interface inheritance, by allowing multiple interfaces to be realized by an actual class.

### 1.3 The object-oriented software life-cycle

No approach to software development is likely to survive unless it solves some of the real problems encountered in software engineering practice. In this section we will examine how the object-oriented approach is related to the conceptions of the life-cycle of software and what factors may motivate the adoption of an object-oriented approach to software development.

Despite some variations in terminology, there is a generally agreed-on conception of the various phases in the development of a software product. Roughly, a distinction can be made between a phase of *analysis*, which aims at specifying the requirements a product must meet, a phase of *design*, which must result in a conceptual view of the architecture of the intended system, and a phase of *implementation*, covering coding, testing and, to some extent, also maintenance activities. See slide ??.

No such consensus exists with respect to the exact relation between these phases. More specifically, there is a considerable variation in methods and guidelines describing how to make the transition from one phase to another. Another important issue is to determine what the products are exactly, in terms of software and documentation, that must result from each phase.

The traditional conception of the software life-cycle is known as the *waterfall*

**The software life-cycle**

- Analysis – Conceptual Model, System Requirements
- Design – System Design, Detailed Design
- Implementation – Coding, Testing

*With an increase in the number of software products not satisfying user needs, prototyping has become quite popular!*

1-19

Slide 1-19: The software life-cycle

*model*, which prescribes a strictly sequential transition between the successive phases, possibly in an iterative manner. Strict regulations with respect to validation of the products resulting from each phase may be imposed to avoid the risk of backtracking. Such a rigid approach, however, may cause severe problems, since it does not easily allow for modifying decisions taken earlier.

One important problem in this respect is that the needs of the users of a system may change over time, invalidating the requirements laid down in an earlier phase. To some extent this problem may be avoided by better techniques of evoking the user requirements in the analysis phase, for instance by developing a prototype. Unfortunately, the problem of accommodating changing user needs and adapting to changing circumstances (such as hardware) seems to be of a more persistent nature, which provides good reason to look at alternative software development models.

**Software development models** The software engineering literature abounds with descriptions of failing software projects and remedies proposed to solve the problem of software not meeting user expectations.

User expectations may be succinctly characterized by the RAMP requirements listed in slide ?? . Reliability, adaptability, maintainability and performance are not unreasonable demands in themselves. However, opinions on how to satisfy these criteria clearly diverge.

**Requirements – user needs**

- Reliability – *incremental development, reuse, synthesis*
- Adaptability – *evolutionary prototyping*
- Maintainability – *incremental development, synthesis*
- Performance – *incremental development, reuse*

1-20

Slide 1-20: Requirements – RAMP

[Bersoff91] and [Davis88] explain how the choice of a particular software development model may influence the chances of successfully completing a software project. As already mentioned, *rapid throwaway prototyping* may help to evoke

user needs at an early stage, but does not help much in adapting to evolving user requirements. A better solution in this respect is to adopt a method of *evolutionary prototyping*. Dependent on the technology used, however, this may cause severe problems in maintaining the integrity and robustness of the system. Less flexible but more reliable is an approach of *incremental development*, which proceeds by realizing those parts of a system for which the user requirements can be clearly specified.

Another means of adapting to changing user requirements is to use a technique of *automated software synthesis*. However, such an approach works only if the user requirements can be formalized easily. This is not always very likely, unless the application domain is sufficiently restricted. A similar constraint adheres to the *reuse of software*. Only in familiar application domains is it possible to anticipate how user requirements may change and how to adapt the system appropriately. Nevertheless, the reuse of software seems a very promising technique with which to reduce the cost and time involved in software products without (in principle) sacrificing reliability and performance. See slide ??.

**Software development models**

1-21

- rapid throwaway prototyping – *quick and dirty*
- incremental development – *slowly evolving*
- evolutionary prototyping – *evolving requirements*
- reusable software – *reduces cost and time*
- automated software synthesis – *one level of abstraction higher*

Slide 1-21: Software development models

Two of the early advocates of object-oriented technology, Cox and Meyer, regard the reuse of software as the ultimate solution to the software crisis. However, the true solution is in my opinion not so straightforward. One problem is that tools and technologies are needed to store and retrieve reusable components. That simple solutions do not suffice is illustrated by an anecdote reported by Alan Kay telling how difficult it was to find his way in the Smalltalk class structure after a significant change, despite the browsing facilities offered by the Smalltalk system.

Another problem lies in the area of human factors. The incentives for programmer productivity have too long been directed at the number of lines of code to make software reuse attractive. This attitude is also encouraged in universities. Moreover, the reuse of other students' work is usually (not unjustifiably) punished instead of encouraged.

However, having a sufficiently large store of reusable software at our disposal will allow us to build software meeting the RAMP requirements stated above, only if we have arrived at sufficiently stable abstractions of the application domain.

In the following, we will explore how object-oriented technology is motivated by problems occurring in the respective phases of the software life-cycle and how

it contributes to solving these problems.

### 1.3.1 Analysis

In academic environments software often seems to grow, without a clear plan or explicit intention of fulfilling some need or purpose, except perhaps as a vehicle for research. In contrast, industrial and business software projects are usually undertaken to meet some explicit goal or to satisfy some need.

One of the main problems in such situations, from the point of view of the developers of the software, is to extract the needs from the future users of the system and later to negotiate the solutions proposed by the team. The problem is primarily a problem of *communication*, of bridging the gap between two worlds, the world of domain expertise on the one hand and that of expertise in the craft of software development on the other.

In a number of publications (Coad and Yourdon, 1991a; Wirfs-Brock *et al.*, 1990; and Meyer, 1988) object-oriented analysis has been proposed as providing a solution to this problem of communication. According to [CY90], object-oriented techniques allow us to capture the system requirements in a model that directly corresponds with a conceptual model of the problem domain. See slide ??.

#### Object-Oriented Analysis

- *analysis = extracting the needs*

The problem domain – **complex reality**

Communication – *with domain experts*

Continual change – *user requirements*

Reuse – *of analysis results*

1-22

Slide 1-22: Object-oriented analysis

Another claim made by proponents of OOP is that an object-oriented approach enables a more seamless transition between the respective phases of the software life-cycle. If this claim is really met, this would mean that changing user requirements could be more easily discussed in terms of the consequences of these changes for the system, and if accepted could in principle be more easily propagated to the successive phases of development.

One of the basic ideas underlying object-oriented analysis is that the abstractions arrived at in developing a conceptual model of the problem domain will remain stable over time. Hence, rather than focusing on specific functional requirements, attention should be given to modeling the problem domain by means of high level abstractions. Due to the stability of these abstractions, the results of analysis are likely candidates for reuse.

The reality to be modeled in analysis is usually very complex. [CY90] mention a number of principles or mechanisms with which to manage complexity. These show a great similarity to the abstraction mechanisms mentioned earlier.

Personally, I do not feel entirely comfortable with the characterization of the analysis phase given by [CY90], since to my mind user needs and system

requirements are perhaps more conveniently phrased in terms of functionality and constraints than in terms of a model that may simultaneously act as an architectural sketch of the system that is to be developed.

However, I do agree with [CY90], and others, that the products of analysis, that is the documents describing user needs and system requirements, should as far as possible provide a conceptual model of the domain to which these needs and requirements are related.

Actually, I do consider the blurring of the distinction between analysis and design, and as we will see later, between design and implementation, as one of the attractive features of an object-oriented approach.

**Analysis methods** The phases of *analysis* and *design* differ primarily in orientation: during analysis the focus is on aspects of the problem domain and the goal is to arrive at a description of that domain to which the user and system requirements can be related. On the other hand, the design phase must result in an architectural model of the system, for which we can demonstrate that it fulfills the user needs and the additional requirements expressed as the result of analysis.

**Analysis methods**

- Functional Decomposition = Functions + Interfaces
- Data Flow Approach = Data Flow + Bubbles
- Information Modeling = Entities + Attributes + Relationships
- Object-Oriented = Objects + Inheritance + Message passing

1-23

Slide 1-23: Analysis methods

[CY90] discuss a number of methods that are commonly used in analysis (see slide ??). The choice of a particular method will often depend upon circumstances of a more sociological nature. For instance, the experience of a team with a particular method is often a crucial factor for success. For this reason, perhaps, an eclectic method combining the various approaches may be preferable (see, for instance, Rumbaugh *et al.*, 1991). However, it is doubtful whether such an approach will have the same benefits as a purely object-oriented approach. See also section ??.

I will briefly characterize the various methods mentioned by [CY90]. For a more extensive description and evaluation the reader is referred to, for example, [Jones90].

The method of *Functional Decomposition* aims at characterizing the steps that must be taken to reach a particular goal. These steps may be represented by functions that may take arguments in order to deal with data that is shared between the successive steps of the computation. In general, one can say that this method is not very good for data hiding. Another problem is that non-expert users may not be familiar with viewing their problem in terms of computation

steps. Also, the method does not result in descriptions that are easily amenable to change.

The method indicated as the *Data Flow Approach* aims at depicting the information flow in a particular domain by means of arrows that represent data and bubbles that represent processes acting on these data.

*Information Modeling* is a method that has become popular primarily for developing information systems and applications involving databases. As a method, it aims at modeling the application domain in terms of *entities*, that may have attributes, and relations between entities.

An *object-oriented* approach to analysis is very similar in nature to the information modeling approach, at least with respect to its aim of developing a conceptual model of the application domain. However, in terms of their means, both methods differ significantly. The most important distinction between *objects*, in the sense of OOP, and *entities*, as used in information modeling, to my mind lies in the capacity of objects to embody actual behavior, whereas entities are of a more passive nature. Concluding this brief exploration of the analysis phase, I think we may safely set as the goal for every method of analysis to aim at *stable abstractions*, that is a conceptual model which is robust with respect to evolving user requirements. Also, we may state a preference for methods which result in models that have a close correspondence to the concepts and notions used by the experts operating in the application domain. With respect to notation UML (the Unified Modeling Language, see Appendix ??) is the obvious choice. How to apply UML in the various phases of object-oriented software construction is an altogether different matter.

### 1.3.2 Design

In an object-oriented approach, the distinction between *analysis* and *design* is primarily one of emphasis; emphasis on modeling the reality of the problem domain versus emphasis on providing an architectural model of a system that lends itself to implementation.

One of the attractive features of such an approach is the opportunity of a seamless transition between the respective phases of the software product in development. The classical waterfall model can no longer be considered as appropriate for such an approach. An alternative model, the *fountain model*, is proposed by [Hend92]. This model allows for a more autonomous development of software components, within the constraints of a unifying framework. The end goal of such a development process may be viewed as a repository of reusable components. A similar viewpoint has originally been proposed by [Cox86] and [Meyer88].

In examining the primary goals of design, [Meyer88] distinguishes between *reusability*, *quality* and *ease of maintenance*. Naturally, reusable software presupposes quality, hence both quality and maintainability are important design goals. See slide ??. In [Meyer88] a rough estimate is given of the shift in effort between the phases of the software life-cycle, brought about by an object-oriented approach. Essentially, these figures show an increase in the effort needed for design. This is an immediate consequence of the observation that the development

**Object-Oriented Design**

- design for maintenance and reuse!

**Software quality**

- correctness, robustness, extensibility, compatibility

**Design projects**

- IDA – Interior Design Assistant
- MASS – Multi-user Agenda Support System

1-24

Slide 1-24: Object-oriented design

of reusable code is intrinsically more difficult. To my mind, there is yet another reason for the extra effort involved in design. In practice it appears to be difficult and time consuming to arrive at the appropriate abstract data types for a given application. The implementation of these structures, on the other hand, is usually straightforward. This is another indication that the unit of reuse should perhaps not be small pieces of code, but rather (the design of) components that fit into a larger framework.

From the perspective of software quality and maintenance, these mechanisms of *encapsulation* and *inheritance* may be characterized as powerful means to control the complexity of the code needed to realize a system. In [Meyer88] it is estimated that maintenance accounts for 70% of the actual cost of software. Moreover, *adaptive maintenance*, which is the adaptation to changing requirements, accounts for a disproportionately large part of the cost. Of primary importance for maintenance, in the sense of the correction of errors, is the *principle of locality* supported by encapsulation, data abstraction and hiding. In contrast, inheritance is a feature that may interfere with maintenance, since it often breaks down the protection offered by encapsulation. However, to cope with changing requirements, inheritance provides both a convenient and relatively safe mechanism.

### Design assignments

Actually designing systems is a complex activity, about which a lot can be said. Nevertheless, to get a good feeling for what is involved in designing a system it is best to gain some experience first. In the remainder of this subsection, you will find the descriptions of actual software engineering assignments. The assignments have been given, in subsequent years, to groups consisting of four or five CS2 students. The groups had to accomplish the assignments in five weeks, a total of 1000 man-hours. That includes formulating the requirements, writing the design specification and coding the implementation. (For the first of the assignments, IDA, C++ was used with the *hush* GUI library. For the second, MASS, Java with Swing was used.) In both cases we allowed for an iterative development

cycle, inspired by a Rapid Application Development (RAD) approach. These assignments will be taken as a running example, in the sense that most examples presented in the book solve in one way or another the problems that may occur when realizing the systems described in the assignments.

**IDA** An *Interior Design Assistant* (IDA) is a tool to support an interior design architect. When designing the interior of a house or building, the architect proceeds from the spatial layout and a list of furniture items. IDA must allow for placing furniture in a room. It will check for constraints. For example placing a chair upon a table will be prohibited. For each design, IDA must be able to give information with respect to pricing and the time it takes to have the furniture items delivered. In addition to the design facilities, IDA must also offer a *showroom* mode, in which the various designs can be inspected and compared with respect to price and delivery time.

1-25

Slide 1-25: IDA

**MASS** An Agenda Support System assists the user in maintaining a record of important events, dates and appointments. It moreover offers the user various ways of inspecting his or her agenda, by giving an overview of important dates, an indication of important dates on a calendar, and (more advanced) timely notification. A Multi-user Agenda Support System extends a simple Agenda Support System by providing facilities for scheduling a meeting, taking into account various constraints imposed by the agendas of the participants, as for example a special event for which a participant already has an entry in his or her agenda. A minimal Multi-user Agenda Support System must provide facilities for registering important dates for an arbitrary number of users. It must, moreover, be able to give an overview of important dates for any individual user, and it must be possible to schedule a meeting between an arbitrary subset of users that satisfies the time-constraints for each individual in that particular group. This minimal specification may be extended with input facilities, gadgets for presenting overviews and the possibility of adding additional constraints. Nevertheless, as a piece of advice, when developing a Multi-user Agenda Support System, follow the KISS principle: Keep It Simple ...

1-26

Slide 1-26: MASS

### 1.3.3 Implementation

In principle, the phase of implementation follows on from the design phase. In practice, however, the products of design may often only be regarded as providing a *post hoc* justification of the actual system. As noted, for instance, in [HOB87], an object-oriented approach may blur the distinction between design and implementation, even to the extent of reversing their actual order. The most important distinction between design and implementation is hence the level of

abstraction at which the structure of the system is described. Design is meant to clarify the conceptual structure of a system, whereas the implementation must include all the details needed for the system to run. Whatever approach is followed, in the end the design must serve both as a *justification* and *clarification* of the actual implementation. Design is of particular importance in projects that require long-term maintenance. Correcting errors or adapting the functionality of the system on the basis of code alone is not likely to succeed. What may help, though, are tools that extract explanatory information from the code.

**Testing and maintenance** Errors may (and will) occur during the implementation as well as later when the system is in operation. Apart from the correction of errors, other maintenance activities may be required, as we have seen previously.

In [Knuth92], an amusing account is given of the errors Knuth detected in the T<sub>E</sub>X program over a period of time. These errors range from trivial typos to errors on an algorithmic level. See slide ??.

**Errors, bugs**

T<sub>E</sub>X

1-27

A – algorithm awry  
B – blunder  
C – structure debacle  
F – forgotten function  
L – language liability  
M – mismatch between modules  
R – reinforcement of robustness  
S – surprises  
T – a trivial typo

Slide 1-27: T<sub>E</sub>X errors and bugs

An interesting and important question is to what extent an object-oriented approach, and more specifically an object-oriented implementation language, is of help in avoiding and correcting such errors. The reader is encouraged to make a first guess, and to verify that guess later.

As an interesting aside, the T<sub>E</sub>X system has been implemented in a language system called Web. The Web system allows one to merge code and explanatory text in a single document, and to process that document as either code or text. In itself, this has nothing to do with object orientation, but the technique of documentation supported by the Web system is also suitable for object-oriented programs. We may note that the *javadoc* tool realizes some of the goals set for the Web system, for Java.

**Object-oriented language support** Operationally, *encapsulation* and *inheri-*

*tance* are considered to be the basic mechanisms underlying the object-oriented approach. These mechanisms have been realized in a number of languages. (See slide ?? . See also chapter 5 for a more complete overview.)

Historically, Smalltalk is often considered to be the most important object-oriented language. It has served as an implementation vehicle for a variety of applications (see, for instance, Pope, 1991). No doubt, Smalltalk has contributed greatly to the initial popularity of the object-oriented approach, yet its role is being taken over by C++ and Java, which jointly have the largest community of users. Smalltalk is a purely object-oriented language, which means that every entity, including integers, expressions and classes, is regarded as an object. The popularity of the Smalltalk language may be attributed partly to the Smalltalk environment, which allows the user to inspect the properties of all the objects in the system and which, moreover, contains a large collection of reusable classes. Together with the environment, Smalltalk provides excellent support for fast prototyping. The language Eiffel, described by [Meyer88], may also be considered as a pure object-oriented language, pure in the sense that it provides classes and inheritance as the main device with which to structure a program. The major contribution of Eiffel is its support for correctness constructs. These include the possibility to specify pre- and post-conditions for methods, as well as to specify a *class invariant*, that may be checked before and after each method invocation. The Eiffel system comes with a number of libraries, including libraries for graphics and window support, and a collection of tools for browsing and the extraction of documentation. The C++ language (Stroustrup, 1991) has a somewhat different history. It was originally developed as an extension of C with classes. A primary design goal of C++ has been to develop a powerful but efficient language. In contrast to Smalltalk and Eiffel, C++ is not a pure object-oriented language; it is a *hybrid* language in the sense that it allows us to use functions in C-style as well as object-oriented constructs involving classes and inheritance.

The newest, and perhaps most important, object-oriented language around is Java, which owes its popularity partly to its tight connection with the Internet. Java comes with a virtual machine that allows for running Java programs (applets) in a browser, in a so-called sandbox, which protects the user from possibly malicious programs. As the final language in this brief overview, I wish to mention the distributed logic programming language DLP (see Eliëns, 1992). The DLP language combines logic programming with object-oriented features and parallelism. I mention it, partly because the development of this language was my first involvement with OOP. And further, because it demonstrates that other paradigms of programming, in particular logic programming, may be fruitfully combined with OOP. The language DLP provides a high level vehicle for modeling knowledge-based systems in an object-oriented way. A more extensive introduction to the Smalltalk, Eiffel, C++, Java and DLP languages is given in the appendix.

**Smalltalk** – *a radical change in programming*

- rapid prototyping

**Eiffel** – *a language with assertions*

- correctness

**C++** – *is much more than a better C*

- the benefits of efficiency

**Java** – *the dial-tone of the Internet*

- security

**DLP** – *introduces logic into object orientation*

- development of knowledge-based systems

1-28

languages

Slide 1-28: Object-oriented languages

## 1.4 Beyond object-orientation?

No introduction to object orientation is complete without an indication of the trends and technologies that surround the field. The word trend should be understood in its positive meaning of set examples and emerging guidelines. And ‘technologies’, such as for example CORBA (the OMG Common Object Request Broker Architecture), as those that set the technological landscape which determines whether object-oriented approaches can be deployed effectively in practice.

**Trends** – *modeling*

- *patterns* – examples of design
- UML – Unified Modeling Language

**Technologies** – *components*

- Web – global infrastructure
- CORBA/DCOM – the software bus
- Java – the platform?

**Challenges**

- Applications → Frameworks ← Patterns

1-29

Slide 1-29: Trends and technologies

At the design front, we may observe two dominant trends. The first may be called the *patterns* movement, which came into the forefront after the publication of *Design Patterns*, authored by a group of authors that is commonly known as the ‘*Gang of Four*’, [GOF94]. The design patterns published there, and elsewhere e.g. [Coplien95], may be regarded as the outcome of mining actual framework and application designs for valid solutions that may be generalized to broader classes of problems. Design patterns focus on understanding and describing structural and behavioral properties of (fragments of) software systems. Equally focused on understanding structure and behavior, but more from a modeling perspective, is the Unified Modeling Language (UML), which has resulted from a common effort of leading experts in object-oriented analysis and design, Grady Booch, Ivar Jacobson and James Rumbaugh, also known as ‘*The Three Amigos*’. UML, indeed the second trend, aims at providing the full notational repertoire needed for modeling every conceivable structural and behavioral aspect of software systems. An excellent introduction to UML is given in [Fowler97]. In Appendix ?? you will find a brief introduction to the UML. With respect to technology, the field is still very much in flux. A dominant factor here is the rapid increase in Internet usage and, more in particular, the Web. The Web has boosted the interest of the IT business world in the deployment of distributed object or component technology to extend their range of business. Nevertheless, the very existence of this infrastructure is in itself somewhat embarrassing, in that the Web and the technology around which it is built is *not* object-oriented. Perhaps it should be, but it simply isn’t. Our embarrassment is aggravated when we observe, following [Szyperski97], that the technology which may change this, in casu component software, is in itself not object-oriented but, paraphrasing the subtitle of this excellent book, ‘*beyond object orientation*’. And even worse, object-oriented approaches at framework development have failed more often than they have succeeded, an observation which is confirmed by for example [Surviving]. Reading this you may think that object-orientation is in a deplorable state, and close the book. It isn’t. First of all, because in terms of modeling and design there is *no* beyond object-orientation. And secondly, quoting Szyperski, ‘object-technology, if harnessed carefully, is possibly one of the best ways to realize component technology ...’. Well, believe me, it is the best way. Whether it is CORBA, Microsof (D)COM or Java that will become the dominant component technology is quite another issue; component technology that ignores the object-lessons is doomed to fail!

**Challenges** Ignoring the component question for the moment, we may ask ourselves what the major challenges are that are confronting us as software developers. Briefly put, we still need to go a long way before we understand our applications well enough in terms of the (problem-solving) patterns underlying their construction that we can realize these patterns robustly in frameworks that are not only reusable conceptually, but that will also be (re)used in practice to develop cost-effective, competitive, economically viable applications. More concretely, a major challenge for the next decade will be to develop and deploy frameworks that operate in areas such as finance, medical care, social welfare and

insurance. This is explicitly not only a technical problem, but also a problem of coming to agreement with respect to the abstractions and corresponding standards that provide the computational infrastructure for these domains. Also on my wish-list is the separation of *logic* and *control*, by which I mean the decoupling of the more or less invariant functionality as may be provided by for example *business objects* and *business processes* and the more variable logic that controls these processes. In other words, it is necessary that the *business logic* is made explicit and that it is factored out of the code effectuating it.

#### Challenges in O-O

1-30

- vertical framework development – finance, medical care, insurance
- separation of 'logic' from 'control' – business rules
- distributed object technology – heterogeneous systems
- visualisation – structure and processes
- knowledge intensive applications – declarative
- heterogeneous systems – fragmented applications

Slide 1-30: Challenges

Another challenge is to integrate the various technologies into our frameworks and systems. In effect we will see more and more heterogeneous systems, composed of components from a variety of suppliers. These components may be implemented in every conceivable language, and may run on different platforms. How to connect these components in a reliable manner is still an open problem. And more generally, although there are solutions for crossing the various boundaries, the platform boundary and the language boundary, there are still a lot of problems to solve. In this book we will explore some of these problems, and get some experience with some of the solutions. Both our hardware and software technology are improving rapidly. Yet, we are still stuck with the WIMP interfaces. In my opinion, it is time for a change. What I would like to see is an exploration of 3D user interfaces and 3D visualisations of the structure and processes underlying information-intensive applications. Although not specifically related to object-oriented software development, this is an area where object orientation can prove its worth. When we think about real applications, for example information or business services on the Internet, they are usually the kind of applications that we may characterize as knowledge-intensive applications. In a somewhat idealistic vision, we may think of application development that consists of composing components from perhaps even a number of frameworks, so that we don't have to bother with the tiresome details of network access and GUI development. Then what remains to be done is to glue it all together, and provide the information and knowledge that enables our application to deliver its services. Partly we can rely on database technology for the storage and retrieval of information. But in addition we will need other declarative formalisms for

expressing, for example, our business logic or, as another example, for expressing the synchronisation constraints of our multimedia presentation.

Considering Web applications, even as they are today, we see applications that consist of a mixture of code, tools and information. The phrase *fragmented applications* seems apt here. For example a store selling books on the Internet needs everything ranging from Javascript enabled webpages, to a secure CORBA-based accounting server. It is very likely that such applications will be developed partly by composing already existing components. In his book, [Szyperski97] argues that component-technology must be considered as the next stage, that is (as the subtitle of his book indicates) *beyond object orientation*. This is true to the extent that naive object orientation, characterized by weak encapsulation and white-box or implementation inheritance, has proven to be not entirely successful. What we need is a more robust specification of the behavioral properties of objects, for example by contractual specifications, and a stronger notion of encapsulation, in which not only the inner world of the object is protected from invasions from the outside, but where the outer world is also shielded from the object itself, so that the object cannot reach out into a world that might not even exist. More concretely, objects must be designed that allows them to be used in a distributed environment. They must observe, as Wegner puts it, the *distribution boundary*.

## Summary

This chapter has given an outline of the major theme of this book, which may be characterized as the unification of a software engineering perspective and a foundational approach. The minor theme may be characterized by saying that a considerable amount of technology is involved.

<b>Themes and variations</b>	1	1-31
<ul style="list-style-type: none"> <li>• <i>terminology</i> – all phrases</li> <li>• <i>object computation</i> – message passing</li> <li>• <i>contracts</i> – for constructing and validating software</li> </ul>		

Slide 1-31: Section 1.1: Themes and variations

In section 1 we looked at the terminology associated with object orientation, we studied the mechanisms underlying object computation and we discussed an approach to the development of software that centers around the identification of responsibilities and the definition of abstract data types embodying the mutual responsibilities of a client and a server object in terms of a *contract*. See slide ??.

Then, in section 2, we looked at object-orientation as a paradigm of programming, extending an abstract data type approach with support for the organization of object types in a polymorphic type structure. See slide ??. Further, an overview

**Paradigms of programming** 2

- *styles of programming* – as a family of conventions
- *data abstraction* – and its possible realizations
- *polymorphism* – and the features of inheritance

1-32

Slide 1-32: Section 1.2: Paradigms of programming

was given of the literature available on OOP, including a number of landmark papers on which this book was originally based.

**The object-oriented software life-cycle** 3

- *software development models* – in particular the role of prototyping
- *software quality* – in relation to reuse and maintenance
- *programming languages* – the choice of a vehicle

1-33

Slide 1-33: Section 1.3: The object-oriented software life-cycle

In section 3 we looked at the object-oriented software life-cycle, consisting of the phases of analysis, design and implementation. We discussed software development models and the role of prototyping, how an object-oriented approach may promote software quality and facilitate maintenance, and we looked at some programming languages as vehicles for the implementation of object-oriented code. See slide ??.

**Beyond object orientation?** 4

- *modeling* – patterns, UML
- *components* – CORBA, (D)COM, Java
- *heterogeneous systems* – separating logic and control

1-34

Slide 1-34: Section 1.4: Trends and technologies

In section 4 we attempted to discern trends in the research and deployment of object-oriented technologies. We also tried to formulate the challenges we are faced with which concern the utilization of components for the development of knowledge-intensive heterogeneous systems, that allow to factor out the (business) logic in a declarative manner. See slide ??.

## Questions

1. How would you characterize OOP and what, in your opinion, is the motivation underlying the introduction of OOP?
2. Characterize the most important features of OOP.
3. Explain the meaning of the phrase ‘*object orientation reduces the complexity of programming.*’
4. How would you characterize *contracts*? Why are *contracts* important?
5. How is OOP related to programming languages?
6. What classes of languages support OOP features? Explain.
7. What influence is an object-oriented approach said to have on the software life-cycle? What is your own opinion? Discuss the problem of maintenance.
8. How would you characterize *software quality*?
9. Mention a number of object-oriented programming languages, and give a brief characterization.
10. What do you see as the major challenges for research in object orientation?

## Further reading

Nowadays there are many books that may serve as a starting point for reading about OO. Dependent on your interest, you may look at [Surviving], which treats issues of OO project management, [Meyer97], which gives an extensive introduction to design by contract and programming in Eiffel, or [Fowler97], which gives a succinct introduction to UML. Alternatively, you may take one of the introductory programming books for Java, from which you will almost certainly learn something about OO as well.