

0.1 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*

0-1

Slide 0-1: 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.

0.1.1 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 ?? . 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 speak

Object terminology

- 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*

0-2

Slide 0-2: 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 ??.

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.

The flexible dispatching behavior of objects that lends objects their polymorphic behavior is due to the dynamic binding of methods to messages. Polymorphic

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

0-3

Slide 0-3: Features of OOP

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 ?? . 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.

OO = encapsulation + inheritance

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

0-4

Slide 0-4: Benefits of OOP

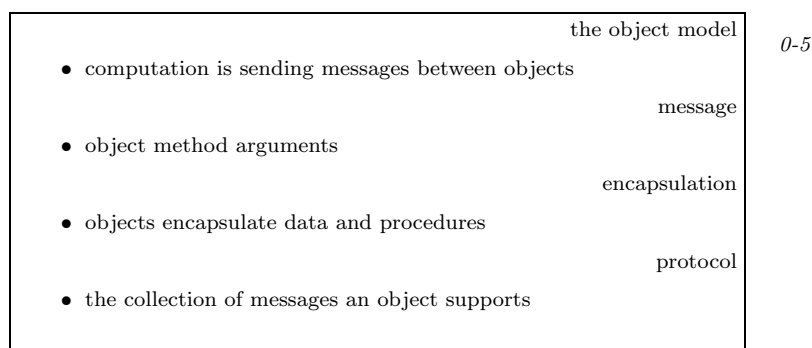
Although an object-oriented approach to program development indeed offers

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.

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.

0.1.2 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.



Slide 0-5: The object model

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 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 ??.

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 computation, 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

0-6

- 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.

Slide 0-6: 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 computable 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 spec-

ification 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.

0.1.3 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 ?? .

Encapsulation

- abstract data types

ADT = state + behavior

Object-oriented modeling

- data oriented

0-7

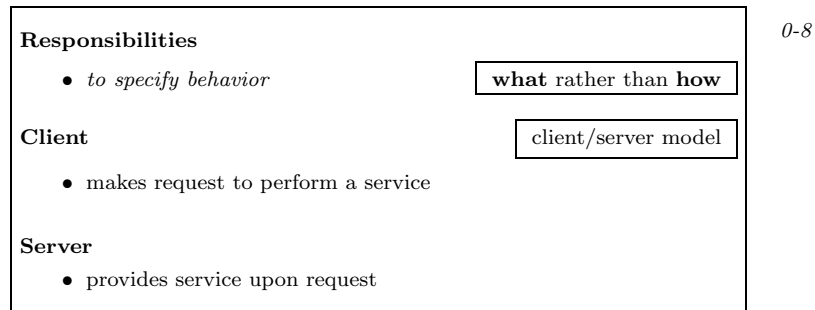
Slide 0-7: Abstract data types – encapsulation

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 ?? .

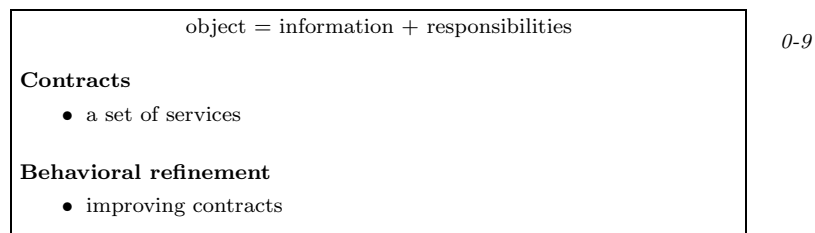
Naturally, the responsibilities of an object cannot be determined by viewing the object in isolation. In actual systems, the functionality required is often dependent on complex interactions between a collection of objects that must cooperate in order to achieve the desired effect. However, before trying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



Slide 0-8: Responsibilities in OOP

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 0-9: 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

0-10

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

Slide 0-10: 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 ??.

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,

attributes <ul style="list-style-type: none"> • more information services <ul style="list-style-type: none"> • better services contracts <ul style="list-style-type: none"> • more and better services A better service <ul style="list-style-type: none"> • fewer restrictions for the client • more obligations for the server 	0-11
---	------

Slide 0-11: Improving services

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.

Object-oriented modeling <ul style="list-style-type: none"> • prototyping, specification, refinement, interactions <p style="text-align: right;">OOP = Contracts + Refinements</p>	0-12
--	------

Slide 0-12: 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 ??.