Logics for OO information systems: a semantic study of object orientation from a categorial substructural perspective

de Haas, E.

Citation for published version (APA):
Chapter 2

Concepts in object orientation

"Object-oriented programming is a wonderful example of how fruitful things don't happen very precisely"

(Robin Milner)

One conjecture of this thesis states that the concepts which are actually used in the practice of object oriented information systems are not similar to the concepts that are common in contemporary mathematics. In contrast, the concepts of information systems evolved from the need and "way of looking at things" in practice. This entails that these concepts do not have a rigorous mathematical definition. There exists, however (and fortunately) quite some level of consensus on the meaning of those concepts (although hardly mathematically defined).

In this chapter we analyze and exemplify the concepts and notions for which we will construct a thorough mathematical theory in the succeeding chapters. We believe that the concepts we captured mathematically are amongst the most pronounced and widely used (and interesting) concepts that have become important in the field of object oriented information systems. Our attempt is to capture at least the common part of the conceptual world of object technology.

In this chapter we start to give an account on the sources from which we deduced the concepts mentioned. Thereafter we will present our interpretation of the concepts we strongly believe to be the most basic and interesting.

2.1 Concepts

The notions of object oriented information systems are defined in many different ways. Not only do the definitions differ in the level of formality, the definitions also differ in the level of conceptuality, i.e. in some presentations the notions
are explained using low level concepts of implementation (directions on how to implement the notions), and at other places the same notions are explained at a high conceptual level in terms of their desired behavior (abstracting totally on implementation). For example the use of an OID (object identifier) is more an implementation 'trick' to obtain the ability to distinguish two objects that have the same values, rather than a philosophically justifiable property of an object. Nevertheless it is a property that is used many times to describe the properties of an object. We do recognize the importance to have both the high (conceptual) level and the low (implementation oriented) level descriptions, because both kinds are actually used. Below we strive to strictly separate the low level and high level descriptions. We believe this results in a more clear picture of the concepts we will describe. In this section we informally describe and analyze the concepts of interest in this thesis.

2.1.1 Object and object identifier

In informal terms, an object in an information system represents some 'actual' entity, whether this entity is, for example, a person, or a scalar value. The most important property of an object is that it has an identity. The notion of identity is, although very common in use, quite difficult to understand ([Leeuwen93]). Many philosophers quarreled with this seemingly unproblematic notion, and, fortunately solved many paradoxes involving the use of identity (e.g. [Frege1892]). In information systems where object identity is important, there is also some quarrelling ([EmdeBoas96]). This quarrelling, though, hardly takes into account the research (in philosophy) on the notion of identity that has already been done.

To avoid many paradoxes when talking about identity, one can make (as in philosophy) a distinction between 'language' and 'meta language'. The meta language contains expressions that reflect directly things in the real world. In the 'language' one can talk about the real world, and expressions in this language are interpreted via this meta language.

In the object oriented information system practice both 'language' and 'meta language' is developed when building a model of the real world. For example rows in a database form pieces of the 'meta language vocabulary', while in some input screen one can type in 'language' expressions to specify a query. However in the object oriented practice no distinction is made between 'language' and 'meta-language' when talking about the real world. Although this is not really harmful, it requires a lot of accurate administration to avoid running into non-clarities and paradoxes. Especially because the concept 'object' and 'identity' is such an important ingredient of the paradigm. Below we sketch out some of the problems.

In (more) formal definitions of objects in information systems, an object identifier (OID) is usually associated with an object. Frequently an OID is some
unrelated value, which is used to distinguish objects for which we may have the same data in our information system, but which are known to be different\(^1\); and also for identifying an object of which some of its data is changed in the course of time. Furthermore an OID often serves as a handle to refer to an object, or in other words it acts as a *name or reference* for the object of which it is the identifier.

### 2.1.1. EXAMPLE

Imagine we have a bag with three marbles, one white marble and two black marbles. We can model this information as follows:

<table>
<thead>
<tr>
<th>marbles</th>
<th>OID</th>
<th>COLOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x123</td>
<td>black</td>
</tr>
<tr>
<td></td>
<td>x456</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>x789</td>
<td>black</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bags</th>
<th>OID</th>
<th>CONTAINS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b789</td>
<td>x123, x456, x789</td>
</tr>
</tbody>
</table>

Note that we used the OIDs of the marbles to denote which marbles are in the bag.

**Question:** Suppose we pick blindfolded a marble from the bag and this marble happens to be black. Which marble did we pick, x123 or x789? If the marbles 'look' exactly the same, does it matter which marble we picked? And what after someone told you that one of the black marbles is cursed?

The 'trick' of using OIDs to solve the matters of identity usually works out fine. Sometimes, though, it does not. For example, if we want to be able to reason with an infinite set of objects, like the natural numbers, we have to assign an OID to all the natural numbers. As this is generally not possible we get the unnatural situation that some natural numbers -the ones we have stored somewhere in the information system- have an OID, and most others, -the ones we have not used yet- do not. To solve this situation most of the systems apply another trick: among the objects they distinguish so called *literals*, which are 'objects without an OID that are identified by their value' (e.g. [Cattell94]). Although this solves the problem of infinite sets of OIDs, the resulting non-uniformity of the collection of all objects is very inelegant and on a conceptual level, even incorrect\(^2\). In

\(^1\)Using this trick, an attempt is made to satisfy Leibniz principle that states that no two individuals can be the same in all properties, without actually being the same. However the property that is introduced to accomplish this (the OID) is without any independent meaning.

\(^2\)all objects have identity, but some objects have more identity than other objects'
our opinion using OIDs is a way to solve the matter of implementing the matter of object identity, but it should not be confused with concepts that handle the notion of identity semantically.

The notion of OID becomes really problematic if we consider incomplete information on the identity of objects or information that possibly contains wrong identifications. In some cases we may not know whether two objects are the same or not. The requirement of associating an OID to an object will usually assign different OIDs to these two objects. But if we discover in due time (by additional information) that these two objects are actually the same, we will have to identify the two objects. How to do this with OIDs is unclear: do we keep both OIDs, or only one of them, do we assign a new OID. Again we argue that the OIDs may implement the notion of identity correctly, but as long as we cannot actually identify two different persons (in reality) this notion is vague, sloppy, and incorrect from a philosophical point of view. There is an alternative way to do it: come up with a philosophically sound 'trick' to handle identity, which is strictly separated from the tricks used to implement this notion. This gives a clear view on the matter of identity of objects, and enables one to reason about identity of objects without encountering disturbing paradoxes or inconsistencies.

We propose the following approach to handle objects and their identity. In the languages we use to talk about the objects in an information system, we have names for the objects. A name is simply a word in the language used, and is interpreted to refer to the actual object it denotes. The most important difference with an OID is that a denotation is not a property of an object. We must take care that different objects have different names. In contrast with the OID approach we do allow one object to have several different names. In order to keep track of the identities of the objects we denote, we maintain an identity relation ($\approx$) between the names and/or a difference relation ($\neq$). Thus if we know that two names $a$ and $b$ denote the same actual object, we will put $a \approx b$. If we know that they denote different objects we put $a \neq b$.

2.1.2. EXAMPLE. Consider again the bag of marbles of example 2.1.1. We will use names, instead of OIDs to refer to an object.

$$\begin{array}{c}
\text{bags} \\
\hline
\text{b denotes the bag containing } m_1, m_2 \text{ and } m_3
\end{array}$$

---

3 Assuming that the identity is an immutable property of an object. Note however this assumption is not totally unquestioned (even outside 'science fiction').

4 We also do not want to reinvent the fruits of 2000 years of philosophy, we simply take a Frege style approach ([Frege1892]) to handle the matter of identity. It is not the switch of philosopher (from Leibnitz to Frege) that bears fruit here, but the fact that we apply the principle of Frege in a semantically correct manner, instead of forcing the Leibnitz principle with philosophically doubtful tricks.

5 It is, roughly, the way mathematicians deal in a language with the identity of the mathematical objects.
2.1. Concepts

Fig. 2.1: Denoting marbles

The diagram illustrates the denotation of marbles by their names.

\[
\begin{array}{c|c}
\text{marbles} & \text{name}_1 & \text{denotes} & \text{black marble } m_1 \\
\text{name}_2 & \text{denotes} & \text{white marble } m_2 \\
\text{name}_3 & \text{denotes} & \text{black marble } m_3 \\
\hline
\text{inequality} & \text{name}_1 \not= \text{name}_2 \\
& \text{name}_1 \not= \text{name}_3 \\
& \text{name}_2 \not= \text{name}_3 \\
\end{array}
\]

(Take symmetric closure of the below)

Note that for simply modeling the structure this approach hardly differs from the one taken in example 2.1.1. The difference in approach becomes apparent if we want to determine identities\(^6\).

Let us now pick a black marble from the bag. From its appearance we cannot determine whether we took out the marble we denoted with \text{name}_1 or the marble we denoted by \text{name}_3. We can only say that we now have a black marble out of

\[\text{merely to know that a name has as its referent an object with which we are confronted, or which is presented to us in some way, at a particular time, is not yet to know what object the name stands for: we do not know this until we know, in Frege's terminology, 'how to recognize the object as the same again', that is, how to determine, when we are later confronted with an object or one is presented to us, whether or not it should be taken to be the same object}^{[\text{Dummett73}]}.\]
the back (in the picture denoted by \texttt{blackm}_{out}), and a black marble in the bag (denoted by \texttt{blackm}_{in}), and that they are different (i.e. \texttt{blackm}_{in} \neq \texttt{blackm}_{out}). Using fixed identifiers would cause problems here.

This approach solves the problems with OIDs we mentioned above. If we want to talk about literals -objects that are nothing more than an immutable value-for example natural numbers, we simply take the token of that value as the name of the object. For example, for the immutable object 'the natural number 1', we take the name '1' or 'one', and for 'the natural number 2' we can take the name '2' or 'two' or 'b10'. This avoids the administration of superfluous OIDs. Also it solves the asymmetry of having object with or without OIDs, because all objects have their names, both non-literals and literals. When an object is changing some of its properties dynamically, its name still remains a valid reference to the object. When in course of time we discover that two names, \(a\) and \(b\) actually denote the same object, we only have one unambiguous action to take: add the equality \(a \approx b\) to our equality relation.

When we analyze this phenomenon, especially the fact that we introduce an explicit (non) identity relation to handle the identity matter, we tend to conclude that this matter reveals a very strong intuition about objects that is neglected when making things precise. Drawing the three marbles from our example (or writing them down) as is done in the object oriented way of looking at things, we implicitly mean to give a lot more information than just labeling objects and giving them a color. Not only do we say that we refer to three objects which are marbles, and of which one is white and two are black; i.e. in logical notation:

\[ \exists x, y, z (\text{marble}(x) \land \text{marble}(y) \land \text{marble}(z) \land \text{black}(x) \land \text{white}(y) \land \text{black}(z)) \]

but implicitly we also mean that these objects are different and that we really have exactly three objects in this bag and also that being white disqualifies being black; i.e.

\[ \exists x, y, z (\text{marble}(x) \land \text{marble}(y) \land \text{marble}(z) \land \text{black}(x) \land \text{white}(y) \land \text{black}(z) \]
\[ \land x \neq y \land x \neq z \land y \neq z \]
\[ \land \forall u [\text{marble}(u) \rightarrow u \approx x \lor u \approx y \lor u \approx z] \]
\[ \land \forall u [\text{black}(u) \rightarrow \neg \text{white}(u)] \]
\[ \land \forall u [\text{white}(u) \rightarrow \neg \text{black}(u)] \]

To strengthen our case, we remark that when we pick a black marble, we really cannot infer from the logical rules which one we picked, while if we had picked the white one, we could really infer from the logical rules that we picked the \(y\) marble (in the scope of that quantor) and from that we could indeed determine its name; i.e.

\[ \forall x, y \text{marble}(x) \land \text{marble}(y) \land \text{white}(x) \land \text{white}(y) \rightarrow x \approx y \]
2.1. Concepts

A modern practical language for analysis and design like UML, which operates on a conceptual level, has circumvented this problem quite wisely by only requiring the existence of a notion of identity that is strong enough to distinguish different objects, leaving it to the implementer of the system how to realize it (in the extensions of the information specification). It also circumvented the mathematical obligation to 'realize' the identity of the objects on a high level in a model, because this language has no formal semantics. This omission, however, can from a practical point of view be justified, because the realizations of UML will not be formal either and moreover will probably use tricks like the OID (depending on the programming/database language used), so the axiom of identity normally suffices to be clear enough about this matter. The challenge is more that the realizations must be such that the problems that occur by using the tricks like the OID are to be solved. A formal model that realized this matter elegantly may provide a good example of how to realize it properly.

2.1.2 Complex value, type and class

Objects can be classified according to their type. Informally a type can be seen as a collection of all objects that have a certain property in common; e.g. the property of being a person or the property of being able to jump, which includes a horse, a flea, and even a person. A common way to classify objects in information systems is to distinguish objects according to their signature. A signature describes what type of basic building blocks an object of this signature should at least possess and which abilities it should at least have and to which other objects it is associated. Given a set of so called basic types, one can build complex types or signatures using several type constructors; e.g.

2.1.3. Example. The signature of the type Book

<table>
<thead>
<tr>
<th>type</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature</td>
<td>attr title : string</td>
</tr>
<tr>
<td></td>
<td>attr author(s) : SET OF(person)</td>
</tr>
<tr>
<td></td>
<td>attr ISBN : N x N x N x N</td>
</tr>
<tr>
<td></td>
<td>attr publisher : company OR institute</td>
</tr>
<tr>
<td></td>
<td>attr year : year &gt; 1450</td>
</tr>
</tbody>
</table>

In the above example we used a couple of common type constructors. Firstly the title part of a book (hence forward called an attribute of book) should be of

\footnotesize

7 for example using a relation administering identity like the Frege style of dealing with identity [Leeuwen93]

8 or collection of properties

9 We will will consider type requirements to be existential
type string. This type is considered a common 'basic' and 'predefined' type in most information systems. Other basic types are integer, bool, float and blob\(^\text{10}\). Most object oriented languages offer, next to the common predefined types, the ability to introduce new basic types, by defining the extension of that basic type. For example,

### 2.1.4. Example.

\[
\begin{align*}
\text{primary-colors} & := \{\text{red}, \text{yellow}, \text{blue}\} \\
N & := 0|S(N)
\end{align*}
\]

Another common type constructor is the **SET** OF type constructor. In the above example a book has a set of authors, which models the event that a book can have an arbitrary number (including zero) of authors.

A well known type construction in mathematics is taking the Cartesian product (\(\times\)) of existing types. In the example, an ISBN number consists of a row of 4 natural numbers; e.g. 90-351-1372-1\(^\text{11}\). We remark that the constructions of taking objects together like the Cartesian product and the 'is attribute of' constructor are not unproblematic from a 'design' point of view. For example we could have chosen to model the object of type Book to be the Cartesian product\(^\text{12}\) of a title, a set of persons, an ISBN, a publisher and a year object. On the other hand we could have chosen to model an ISBN number to be a complex object with 4 attributes of type N. Although for the Book object and for the ISBN object these are quite clearly not the intuitive ways to model them, for other object types this may be less clear a matter. The distinction that is generally made is that an aggregation operator for taking things together (like the Cartesian product) represents an 'IS A' relation between the object and the whole of the aggregation expression; or in other words a member of the aggregation IS part of the whole object. The attribute construction on the other hand models a 'HAS A' relation between the object and each of its attributes. In the example this means the a Book 'HAS A' title and HAS An ISBN number etc. etc.; while the ISBN number IS An (ordered) aggregation of four natural numbers.

To illustrate that this is a realistic problem we point to the practical programming language C++. Most C++ books (e.g. [Stroustrup91]) explain the attribute constructor (member in C++) as a modeling a 'HAS A' relation to distinguish it from an inheritance relation between classes. For example, if an ISBN number IS An aggregation of four numbers, it should be a subclass of the type \(N \times N \times N \times N\), inheriting its properties. A Book on the other hand HAS

\(^{10}\)blob is an abbreviation of 'binary large object', a type that is much used in multi-media information systems.

\(^{11}\)We realize that the ISBN number encodes some information. For the sake of the example we only consider it as a sequence of numbers.

\(^{12}\)we will call such a construction an aggregation.
2.1. Concepts

A title, meaning that title should be a member of a Book. The new generation information system language UML also acknowledges complexity of the modeling choices that have to be made in this case and proposes next to attribute (HAS A) construction an aggregation operation in two flavors (to provide a 'middle course'). UML has two types of aggregations, relating these types of aggregation to a notion of 'life time dependency'. Between two objects with a lifetime dependency there exists an 'IS PART OF' relation (i.e. between the 'aggregation of all these parts' and the 'object' there exists an IS A relation), while between two objects that have no lifetime dependency there is a HAS A relation. A typical example of this phenomenon is the following: IS a bicycle a frame together with a saddle and wheels and a steer, or HAS a bicycle a saddle, a frame wheels and a steer (see figure 2.2). In the first view the bicycle is not the same anymore from an identity perspective when you replace its front wheel, because the parts that make up the definition of the bicycle changed. In the second view you can replace all its parts (all the parts that are modeled) and still have the same bicycle from an identity perspective.

The 'OR' type constructor in the above Book example is yet another type constructor which is generally known as the union type constructor. In the example, a book can be published by a company (e.g. North Holland publ. company) or an institute (e.g. the Institute of Logic Language and Computation (ILLC)).

It can be the case that for some reason one wants to constrain the set of objects of an existing type to some subset of this set. For example one wants to consider only printed books, and book printing was invented in 1450, so one considers books that were published after 1450. The ability to constrain a type is also a type construction. Although in type theory there is not much difference in defining a type by giving its signature or by formulating constraints, in information systems...
these two activities are often separated, and often totally different languages are used to perform these two ways of defining types. For example the design language UML has many kinds of graphical schema techniques to write down the signature with limited ability to add some constraints. To write down constraints with full expressive capacity, UML has defined a (textual) constraint language OCL ([WarmerKleppe99]).

The concept of a class coincides largely with the concept of type. A class is also used to classify objects according to several properties. Next to properties and constraints, a class also emphases abilities (which for simplicity we also consider to be properties of an object) by separately listing the operations that are associated with the type of objects in the class. Additionally, a class also contains the implementations of the functions (i.e. we can talk about methods) and the manner in which the objects are actually represented in the implementation. In other words a class consists of:

1. type
2. operations
3. body of implementation containing:
   - actual representation of the type of objects
   - implementations of the operations

2.1.5. Example. In this example we see the specification (coding) of a banking account type and its behavior.

class Account {
   // the signature of Account
   int account;
   string owner;
   float balance;

   public:
   // the methods of Account
   Account(int accountnr, string owner, float balance);

   float get_balance();
   void incr_balance(float amount);
   void decr_balance(float amount);

   private:
   // some things needed for implementation
   float my_percent; // the interest percentage
   date prev_mutation_date; // date of previous mutation
2.1. Concepts

```cpp
float build_interest; // to keep track of the interest
```

// The body with the implementation of the constructor of the class
// and the implementations of the methods.

Account::Account(int accountnr, string owner, float balance)
{
    my_percent := NORMAL_PERCENT;
    prev_mutation_date := current_date;
    build_interest := 0.0;
}

float Account::get_balance()
{
    return balance;
}

void Account::incr_balance(float amount)
{
    float add_interest;
    float add_percent;
    add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
    add_interest := balance * add_percent;
    balance := balance + amount;
    build_interest := build_interest + add_interest;
    prev_mutation_date := CURR_DATE;
}

void Account::decr_balance(float amount)
{
    float add_interest;
    float add_percent;
    add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
    add_interest := balance * add_percent;
    balance := balance - amount;
    build_interest := build_interest + add_interest;
    prev_mutation_date := CURR_DATE;
}
```

2.1.3 ISA hierarchy, subtyping and inheritance

Types can be ordered in a type hierarchy or ISA hierarchy. For example consider the two types student and person and suppose one would like to assert that a
student is a (ISA) person, or in other words, one would like to assert that student is a subtype of person. This means that the set of all objects of type student should be contained in the set of all objects of type person. Usually a subtype relation like the one above is defined by adding phrases to the information system that formulate the subtype relation; e.g.

\[
\text{student ISA person}
\]

From a collection of these phrases, other unstated subtype relationships can be inferred. For example, if we also add the phrase graduate-student ISA student, we should be able to infer graduate student ISA person. Usually there is a formal system with inference rules that take care of these things. An influential paper on this subject is from Cardelli ([Cardelli84]). A system for subtyping usually contains, next to rules for reflexivity and transitivity of the ISA relation, the following rule (*):

If type \( A \) contains at least all the attributes that type \( B \) has, then \( A \) is a subtype of \( B \)\(^{13}\)

For example:

2.1.6. EXAMPLE. Look at the following three type definitions.

\[
\begin{align*}
\text{type vehicle} &= \text{(age : integer, speed : integer)} \\
\text{type machine} &= \text{(age : integer, fuel : string)} \\
\text{type car} &= \text{(age : integer, speed : integer, fuel : string)}
\end{align*}
\]

According to the rule (*) we can derive the following relations:

\[
\begin{align*}
\text{car ISA vehicle} \\
\text{car ISA machine}
\end{align*}
\]

In some cases though this inference rule can result in undesired subtype relationships; for example:

2.1.7. EXAMPLE. Consider the following type definition:

\[
\text{type fuelcontainer} = \text{(age : integer, fuel : string, size : measure-of-volume)}
\]

\(^{13}\)The rule I am aiming at is even more general than that stated above. The above rule, though, already suffices for the argument. Usually the rule looks as follows:

If \( A = [c_1 : C_1, \ldots, c_m : C_m, \ldots, c_n : C_n] \) and \( B = [d_1 : D_1, \ldots, d_m : D_m] \) and also \( C_1 \text{ ISA } D_1 \cdots C_m \text{ ISA } D_m \) and \( c_i = d_i (1 \leq i \leq m) \) then \( A \text{ ISA } B \)
Using the rule (*) we can unfortunately derive fuelcontainer ISA machine.

Even more treacherous is the application of rule (*) in the following example. Consider the following two type definitions for a polygon and a polyline respectively,

```
type polygon
  
signature
  attr     pointlist : LIST OF point
```

and

```
type polyline

signature
  attr     pointlist : LIST OF point
```

It is usual to represent both a polyline and a polygon by a sequence of points. But neither is a polyline a polygon nor vice versa. i.e. polyline IS NOT A polygon and polygon IS NOT A polyline

In our model, which we present in the chapters to come, we have chosen to omit the mentioned subtyping inference rule (*). This will give the typing system more freedom. We will discuss this matter when addressing the notion of 'knowledge rules'. It is feasible to regain the possibility to accomplish the subtyping inference from example 2.1.7 when it is desired, by putting things just a little differently; e.g.

**2.1.8. Example.** Consider the type definitions of example 2.1.6. We can define the types for machine, car, and fuel-container with the following phrases:

If an object is a machine then it has an age attribute and a fuel attribute; i.e.

```
machine -> attr(age) ∧ attr(fuel)
```

If an object is a car then it has an age attribute and a speed attribute and a fuel attribute; i.e.

```
car -> attr(age) ∧ attr(speed) ∧ attr(fuel)
```

If we want to be able to derive that a car ISA machine we will have to add the phrase

If an object is a car then it is a machine; i.e.

```
car -> machine
```

If we want the effect of the (*) rule we should formulate the properties of a type a little different; namely:
Chapter 2. Concepts in object orientation

Every object that has a speed attribute and an age attribute is a vehicle;
i.e.
\[ \text{attr}(\text{age}) \land \text{attr}(\text{speed}) \rightarrow \text{vehicle} \]

Now we can derive that a car ISA vehicle without explicitly stating the ISA rule (i.e. we do not derive it from simple rules of reflexivity and transitivity of the ISA relation).

Classes can be ordered in a class hierarchy, similar to the way types are ordered in a type hierarchy. The ordering among classes is given by a so called inheritance relation. In many cases the notion of inheritance largely coincides (in effect) with the notion of subtyping. But again, with inheritance there are matters of implementation that significantly determine the ordering of the classes, where this is not the case with types and subtyping. If we state that a class $B$ inherits the properties of a class $A$, we will say that $B$ is a subclass of $A$. Also, here the objects of class $B$ have all the important properties such that we can view them as objects of its superclass $A$. Some of the properties that are inherited (i.e. properties that determine the inheritance relation) are related to the implementation of the classes, which include the methods and their code and the attributes with their representations. A precise definition of inheritance is hard to give. An elaborate taxonomy article on inheritance results 7 different ways inheritance is used in the literature (see [Tailvalsaari96]).

In most Object Oriented programming languages there is a subtle difference between the use of subclasses versus subtypes. In C++, an instance of a subclass is not seen as an instance of the superclass, while an instance of a subtype is always an instance of the supertype. For example, in C++ if a class circle is a subclass of a class shape, then an instance of circle is not also an instance of shape. In this case it is said that an instance of class circle can play the role of an instance of class shape. The reason for this is related to the interpretation of properties of the whole class like the number of instances of a class.

Furthermore, some implementation matters play a role, like problems of choosing which implementations of methods have to be executed for an instance when one does not know how much further down in the class hierarchy the object may be specified. Object oriented design languages (e.g. UML) explicitly leave room for both the subtype and the subclass interpretations. The user then may choose the interpretation based on the language he will use to realize his designed system.

Even though in effect the concept subclass coincides largely with the concept subtype the main drive for subclassing seems to be based on code sharing. The code of class $A$ is used to implement a large part of its subclass $B$. This is, of course, a typical scenario when $B$ is a subtype of $A$; i.e. when a $B$-object is also an $A$-object. Not all the code of $A$ is always inherited though. In practice it became clear that the most strict notion of inheritance, which proscribes that all
the code of a superclass should be inherited, is not flexible enough to obtain both a high amount of code sharing and nice looking hierarchies. For that reason, in a subclass it is allowed to re-implement a method that is inherited from a superclass (i.e. not inherit the code of that method). This notion is known under the name *overriding*. The ability to override code from an inherited method necessitates another, purely implementation oriented phenomenon, which is that of *dynamic binding*. Traditionally, in the compilation of a program, a function (method) name is uniquely bound to a piece of code, which has to be executed when the function is called by its name. With methods in a class hierarchy this is not possible, because the implementation of a method can be overwritten at some point in the class hierarchy (while the typing of the method stays the same!).

2.1.9. Example. Consider the class definition of example 2.1.5. A subclass of the class Account should be a Savings-account, which is an account that gives more interest relative to the amount that is on the account but which forbids a negative balance and has a limit of how much money you may draw from your account in one withdrawal\(^\text{14}\).

```cpp
class Sav_account: superclass Account {
    float my_decr_limit;
public:
    Sav_account(int accountnr, string owner, float balance, float my_decr_limit);
};

// The body with the implementation of the constructor of the class
// and the implementations of the methods. This class inherits the methods
// show_balance and incr_balance, but has a constructor of itself and overrides
// the decr_balance method of its superclass.

Sav_account::Sav_account(int accountnr, string owner, float balance)
{
    my_percent := HIGHER_PERCENT;
    prev_mutation_date := current_date;
    build_interest := 0.0;
};

void Sav_account::decr_balance(float amount)
{
    float add_interest;
    float add_percent;
    if (amount <= MAX_WITHDRAW && balance - amount >= 0.0)
    {
```

\(^{14}\text{Note again that, for clarity and non-C++ speakers, we do not use pure C++ syntax but a pidgeon OO programming language to declare some class to be a subclass}\)
Chapter 2. Concepts in object orientation

add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
add_interest := balance * add_percent;
balance := balance - amount;
built_interest := built_interest + add_interest;
prev_mutation_date := CURR_DATE;
}; //fi

For this reason, the code of a method can only be bound to its name when the
method is actually called by an object. Only then, from the (sub)class membership
of the object that calls the method, one can deduce which code to execute.
This event is called dynamic binding.

2.1.4 Methods and operations

An important ingredient in current information systems is the ability to add dynamics to the objects in the system. This is done by adding operations to the
information system. These operations alter the information in the information
system and/or produce side effects that exhibit the desired behavior of the in-
formation system. In traditional OO systems these operations are exclusively
associated with a type or class. For example:

2.1.10. Example. For specifying a bank account consider the following type:

\[
\text{type Account} \\
\text{signature account-number} \\
\text{owner} \\
\text{balance}
\]

With the type Account we can typically associate the following operations:

\[
\text{operations} \\
gtext{get-balance} \\
increase-balance(amount) \\
decrease-balance(amount)
\]

For operations that take more then one argument the strict connection between
a type and a operation often amounts to problems of symmetry, also known as
the problem of the cow and the milk-can. Suppose you have an operation \( M \)
that models the event of milking a cow resulting in a filled milk-can\(^{15}\). The

\[^{15}\text{i.e. The value of the milk attribute of the cow decreases with the minimum of the amount of milk that the cow passes and the amount of milk that fits in the milk-can, while the milk attribute of the milk-can increases by the same amount}\]
problem now arises in determining which type we have to associate the milking operation $M$, the cow, or the milk-can, or something else. In other words, do we say to the cow: "Put your milk in the milk-can"; or do we say to the milk-can: "Extract the milk from the cow"; or alternatively, do we create a farmer and say to him: "Extract the milk from the cow and put it in the milk-can". The first two possibilities simply point to the asymmetry of the operation call, which arises when an operation with more than one argument is associated to one type only\footnote{This also is the case if both the cow and the milk-can have their own milking operations}. The last possibility of introducing a farmer solves the asymmetry but introduces dummy objects of dummy classes, i.e. objects that only exists to ensure the symmetry of the operation calls, but do not carry any necessary information\footnote{they only have identity and nothing else}. In mathematical type theory the above case is not considered a problem. The milking operation $M$ is simply associated to the Cartesian product of the types for the cow and the milk-can, i.e. to type $\text{cow} \times \text{milk-can}$. Unfortunately in most database systems that carry the object notion combined with dynamics, this type constructor is not readily available for combining complex types. The constructor is very common in the traditional relational database model, but here we have no object notion and no dynamics. We will consider the ability to combine arbitrary types as an important and basic constructor for types. This way we can simply associate operations with the types of the objects they process.

In summary, an operation in an information system will be associated with a type, and is assumed to (possibly) change the content of the information system performing some combination of the following:

1. Altering the attributes of existing\footnote{i.e. present in the considered information system} objects,
2. deleting existing objects,
3. creating new objects,
4. perform side effects that do not alter the content of the information system.

In OO programming languages and database languages operations are, next to a name and a type, also associated with a specific implementation. Taking the name and the type together with the implementation we obtain what is usually called a method (although in a fairly new language like UML both items are separated; a signature there is called an operation while its implementation is called a method).

2.1.11. Example. Consider the type and methods of example 2.1.10. Note that the 'get-balance' method is actually a one-argument function, taking an account and returning a balance. The methods 'increase-balance' and 'decrease-balance'
are actually two-argument functions, taking an Account and an amount, and returning an Account. i.e.

\[
\begin{align*}
\text{get-balance} & : \text{Account} \rightarrow \text{balance} \\
\text{increase-balance} & : \text{Account} \times \text{amount} \rightarrow \text{Account} \\
\text{decrease-balance} & : \text{Account} \times \text{amount} \rightarrow \text{Account}
\end{align*}
\]

The main restriction in the last two functions (and in all functions that are methods that update the object they are a method from) is that the identity of the Account in the domain and the co-domain is the same. So, in a general setting, the first method can be better viewed as a function of the following signature:

\[
\text{get-balance} : \text{Account} \rightarrow \text{Account} \times \text{amount}
\]

Instances of these methods can be viewed as 'Curry-ed' versions ([Barendrecht84]) of the general function taking an Account from the domain into the function.

### 2.1.5 Encapsulation

Consider an information system for which there exists a nice categorization of the objects in classes. It is necessary to force a programmer that makes an application with this information system to really use this categorization. So instead of giving this programmer access to the representation of the objects, we only allow (her/him) the use of the methods of a class to operate on the objects of this class. From the point of view of the mentioned programmer, the representation of the objects is hidden, and only the method names are visible. This is called encapsulation. Encapsulation is a mechanism that enables the concept of a 'class' to be a real categorization, it makes sure that the category specified by the class is really used as such an informal category or type. In a sense the mechanism of encapsulation enables one to force a behavior of the 'classes' to be abstract (very similar to the behavior of 'types'), because it makes certain that a class is more than a collection of code that can be inherited.

### 2.1.6 Declarativeness

One of the main advantages of relational databases and deductive databases is their declarativeness. Declarative languages have some clear benefits over procedural languages, which are common in most OO databases. In order to query a database with a declarative query language one has to specify what information one wants to obtain from the database, and not how to obtain it.

Not only for a query language, but also for an object data definition language, declarativeness is a desirable, because then one only has to state what the type of data (signature) should be and which constraints the data should satisfy, instead of actually specifying how the data should be represented and how the constraints should be enforced.
2.1. Example. In this example, we specify an electrical circuit with resistances. Instead of telling how the different objects are stored in our information system and telling how the different physical quantities can be obtained from the representation of the objects, we list all the attributes and the physical laws that relate the physical attribute to each other. This means we have the following types (expressed in a semi-logical language):

- \[ \text{circuit} \rightarrow \text{attr}(v) \land \text{attr}(i) \land \text{attr}(r) \]
- \[ \text{serial} \rightarrow \text{attr}_1(\text{circuit}) \times \text{attr}_2(\text{circuit}) \]
- \[ \text{parallel} \rightarrow \text{attr}_1(\text{circuit}) \times \text{attr}_2(\text{circuit}) \]

The first type definition says that a circuit has three attributes: a voltage \( v \), a current \( i \) and a resistance \( r \). The second type signature definition says that a serial circuit has two (serially connected) circuits, and similarly the third definition says that a parallel circuit has two (connected in parallel) circuits. The following constraint says that a circuit is either a resistance or a pair of parallel connected circuits or a pair of serial connected circuits. Note that this specification covers all possible simple circuits of parallel and serial resistances.

\[ \text{circuit} \rightarrow \text{serial} \lor \text{parallel} \lor \text{resistance} \]

We now declaratively specify the physical rules for these simple electrical circuits, i.e. Ohm’s law \( v = i \cdot r \) and the rule for current in parallel circuits\(^{19} \) and for resistance in a serial circuit\(^{20} \):

- \[ \text{circuit} \rightarrow \text{equal}(\text{attr}(v), \text{attr}(i) \cdot \text{attr}(r)) \]
- \[ \text{circuit} \land \text{parallel} \rightarrow \text{equal}(\text{attr}(i), \text{sum}(\text{attr}_1(\text{attr}(i)), \text{attr}_2(\text{attr}(i)))) \]
- \[ \text{circuit} \land \text{serial} \rightarrow \text{equal}(\text{attr}(r), \text{sum}(\text{attr}_1(\text{attr}(r)), \text{attr}_2(\text{attr}(r)))) \]

Now it is possible to store (or have for insert or update) the information on a specific circuit in many different ways that is not complete, while (by inference) one can compute all defined quantities of the circuit. Given a proper constraint solver one does not have to specify how the values have to be computed for every case. For example a constraint solver like RL [Denneheuvel90] could infer from the above rules that for parallel circuits, 1 divided by the resistance of the circuit equals the sum of 1 divided by the resistance of its parts; i.e.

\[
\frac{1}{\text{r}_{\text{tot}}} = \frac{1}{r_1} + \cdots + \frac{1}{r_n}
\]

\(^{19}\)The current in a parallel circuit equals the sum of the currents in its parts.

\(^{20}\)The resistance in a serial circuit equals the sum of the resistances of its parts.
Declarativeness is in our opinion an important key to nice and user friendly information systems. So next generation information systems should support declarative languages for defining and querying data.

Most declarative languages have nicely and mathematically defined relations as the basic structure of the database they talk about. Being a little ahead of events, we remark that in the coming chapters we will build a mathematical structure for complex objects for which we provide a declarative language for both defining and querying an object database\textsuperscript{21}. For defining databases, this language will enable us to state many things that are common in non-declarative OODB languages. The only difference is that those statements are logical (they state what has to be the case) and are not directions towards the implementation or representation.

### 2.1.7 Rules and knowledge

In the preceding chapters we have seen some rules that define constraints on the types of an information system. We mentioned subtyping statements as described in section 2.1.3, like: 'A car is a vehicle'. We have also seen a phrase stating the following constraint: 'Every printed book should have appeared after the year 1450'. We mentioned constraints in section 2.1.2 where we discussed signatures of types. But using a general language, many more complex statements are possible. We could introduce negation and disjunction etc.. For example: 'If a vehicle is older than 100 years, or if it has a propeller, then it is not a car'. Or more complex: all laws of quantum mechanics. What we are aiming at is that in using a language which talks about information systems, it is desirable that this language is expressive enough to state phrases as the one above. This enables one to utilize the information system as a knowledge base. We will be able to check or enforce or even prove complex statements that bear the knowledge of experts in the field in which we use the information system. All of this, of course, depends on the expressiveness of the language in which one can express the constraints and rules.

### 2.1.8 Graphical representation

In the field of analysis and design there exist many graphical techniques to model information systems. The models, written using these graphical techniques, are usually translated to database and programming languages that in turn implement the modeled information system. Some of the graphical techniques from analysis connect closely to certain programming database languages. The most apparent example is the close connection between the (E)ER\textsuperscript{22} model and the

\textsuperscript{21}Remember that this is what this thesis will deliver

\textsuperscript{22}(Extended) Entity Relationship model
2.1. Concepts

languages of the relational model\textsuperscript{23}

2.1.13. EXAMPLE. An example of the tight connection between an ER model and a specification of the ER model in the relational database schema.

The ER schema above relates to the following relational database schema\textsuperscript{24}:

<table>
<thead>
<tr>
<th>PILOT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeno</td>
<td>integer</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>address</td>
<td>string</td>
</tr>
<tr>
<td>qualifications</td>
<td>list-of-airplane-types</td>
</tr>
<tr>
<td>roster-id</td>
<td>integer</td>
</tr>
<tr>
<td>key:</td>
<td>employeeno</td>
</tr>
</tbody>
</table>

\textsuperscript{23}e.g. Relational Algebra, QUEL, SQL, QBF, etc. etc..

\textsuperscript{24}There are a number of ways to turn an ER schema into a relational schema. This one is a common transformation taking into account the cardinality of the associations. For a discussion about such transformations one can look in standard relational database textbooks (for example in [Ullman88]).
Chapter 2. Concepts in object orientation

<table>
<thead>
<tr>
<th>TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskno</td>
</tr>
<tr>
<td>start</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>key:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROSTER-TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>roster-id</td>
</tr>
<tr>
<td>taskno</td>
</tr>
<tr>
<td>foreign key:</td>
</tr>
</tbody>
</table>

The tight connection between the (E)ER syntax and the language of the relation databases initiated actually incorporating the (E)ER graphics in a database definition and query language for relational databases (e.g., look at the language HQL [AndriesEngels94]). The reason for the existence of a seamless transition from the (E)ER language to a language for relational databases lies in the fact that both languages talk about the same basic and mathematically defined structure: the relation. In other words, both the (E)ER model and the relational database model have a clear and rigorously defined mathematical semantics, based on the same mathematical structure called relation.

With the connection between graphical object-oriented or object-based analysis and design languages, and object-oriented or object-based database models the situation is completely different. Not only do these models lack a rigorous mathematical semantics, which alone makes the task of establishing the connection difficult, but there also exists a difference in the level of conceptuality between the notions of analysis and design versus the notions of object oriented or object based databases. Although the notions in both domains are similar, the definitions of these notions in analysis and design are more abstract and oriented towards reasoning (debating) about the information system. The similar notions in object-oriented databases are often oriented towards implementing the structures which formulate the notions.

On the other hand, the languages of object-oriented analysis and design use the notions of new generation information systems and are much more liberal in mixing graphical syntax with textual syntax, thereby improving the usability.

2.1.14. Example. The following UML class diagram with OCL (object constraint language) phrases denotes a slightly more informative model for the pilot roster example in the (E)ER diagram of 2.1.13.
2.1. Concepts

A note on the notation: The underlined header denotes the context of the OCL rule; with the dots (.) one can walk along the associations; and the arrow (→) denotes the reference to a method. In the above rule the expression `self.contains` in the context of `Roster` denotes the ordered list of task objects that can be reached from a Roster object. The expression `self.contains→collect(type)` in turn, denotes the invocation of the `collect` method of the ordered list objects where the recipient of the method is the ordered list of tasks objects that is reachable from a particular Roster object and the argument is `type` (an attribute name; i.e. a string object). This method returns the set of types as an ordered list of tasks. The `self.from.qualifications→asSet` expression denotes the invocation of the `asSet` method on the qualification object that is reachable from a roster object via the pilot object. The rule says that that for all the tasks (in a roster) the pilot (of that roster) should have the necessary qualifications.

In this thesis we aim to provide a mathematical foundation of the basic concepts of the object oriented and object based databases and analysis and design methodologies. The foundation will play the same role as the basic concept of relation in the (E)ER model and the relational database model. We will use this foundation both for interpreting the graphical syntax25 like that which exists in analysis and design, and the non-graphical syntax which is common in the languages of databases. As a matter of fact, in some cases we even prefer the graphical syntax over the non-graphical because it shows more clearly the structure of the entity it denotes. We will also incorporate the graphics into the database languages.

25We need to be more formal about the term 'graphical syntax', because for a real 'syntax' one needs a formal 'syntactic theory'. We will explain our view on graphical syntax in chapter 4 when we do the theory, because this matter really is theory.
2.1.9 Partial specifications, Identity, and the Extendibility principle

The concepts of class and type hierarchy of the object oriented information systems both give, in effect, a way of manipulating entities for which one only needs a partial specification. For example, one can specify rules or actions for a machine without knowing whether this is a car or a generator or whatever machine. In other words, we are able to talk about objects taking into account only a part of its information. In order to talk about a (structural) part of an object we need to distinguish partial descriptions of the specification of an object. In most OO languages one has the ability to write down (talk about) the individual connections between an object and its attribute. These connections are the building blocks for describing structure of an object. For example, if we want to talk about a car or a generator as if it is a machine, we look only at the connections between the object we consider and the objects that describe the machine part of that object. These connections (later on we will name them links) make up part of the specification of the car or generator we generalized to a machine.

Analysing this feature that enables one to look at only a part of the specification, one can generalize its intent and assume that objects have (potentially) only a partial specification. This enables one to consider objects in an information system, for which we, at a certain point in time, or at some level of abstraction, have only partial knowledge, and for which this knowledge can be augmented in course of time or when we de-generalize. The fact that in our models the objects have an identity, and therefore are not identified by their structural and behavioral specification, enables one quite naturally to handle the objects like they have only a partial specification. Furthermore it is very relevant in practice, where often at some point in time there is no complete knowledge about all objects in the information system.

We argue that this incompleteness of the specification is tightly connected to the concept of identity in information systems. The fact alone that we can distinguish an object by its identity without knowing its complete specification gives a lot more strength to the concept of identity. In fact it is the concept that makes the Object Oriented paradigm work. The ability to make powerful generalizations (i.e. consider superclasses) makes the languages that support object orientation very expressive. And furthermore, being able to dynamically discover more specific information about an object, and to classify an object at a finer granularity (by considering the subclass), makes the system very flexible. If we would need a complete specification, like for example in the relational database model, the identity would be nothing more then a label to distinguish it from other objects with the same specification. The concept of identity in a system that allows partial specifications and generalizations enables one to talk about object

\[26\text{i.e. the part of the structure of the object that makes this object a machine.}\]
at all levels of granularity without coming into problems with generalizations and de-generalizations.

The assumed partialness of the specification of an object has an important consequence for the models of information systems, though: in such a model, the composition of all the parts of the object does not result in the object itself, because one never knows whether the specification is complete. We will, in the system we present in the next chapters, introduce the concept of extendibility, and say that a partial specification can extend to the whole object it is part of.

2.2 Summary

In this chapter we analyzed the most advertised concepts of object orientation. In the subsequent chapters we will present a mathematical formalization of a large part of these concepts. We will see a mathematical model for object oriented information that could play a role in clarifying and enhancing object oriented information processing, similar as the relational model does for relational information.