Abstract. Methodologies within software engineering are introduced with the speed of improved soap-powders. One of the major movements is object-orientedness. It already started at language level long ago, with Simula 67 and Smalltalk, and nowadays many methodologies are labelled as object-oriented. And it is ongoing: unifications of object-oriented methods are introduced to get the maximum benefit of all involved. This article examines some important techniques and constructions used in the mainstream object-oriented methodologies, and elaborates on their common properties and weaknesses in practical use.

1 Introduction

“Software engineering” is the application of scientific principles to (a) the orderly transformation of a problem into a working software solution and (b) the subsequent maintenance of that software until the end of its useful life.

Software engineering is more than just programming, it also includes all documentation which is necessary to install, use, develop and maintain these programs. People and projects following an engineering approach generally pass through series of phases, as defined in the process model that is used. Furthermore, additional characteristics of applied software engineering are: software systems are built in teams rather than by individuals and they are made up by technical as non-technical aspects as well. Not only a thorough knowledge of computing science, also the ability to communicate, to plan and to manage are critical success factors in the field of software engineers.

A methodology is an explicit prescription for an activity, or set of activities, as required by the selected approach. A methodology, when applied to the right type of project and used by capable people, will with a high degree of probability, with a predictable amount of resources, lead to a solution of the problem.
The availability of (good) software development methodologies may be an indication for the maturity of the software engineering discipline.

The Capability Maturity Model [1] defines five levels of process maturity of which the lowest level is, euphemistically, called "Initial". The description, however, is less disguising: The Initial Process Level could properly be called ad hoc, and it is often even chaotic. At this stage the organisation typically operates without formalised procedures, cost estimates and project plans. Tools are neither well integrated with the process nor uniformly applied. Change control is lax, and there is little senior management exposure or understanding of the problems and issues. Since many problems are deferred or even forgotten, software installation and maintenance often present serious problems.

The past ten years the Software Engineering Institute (SEI) measured the level of organisations that are active in a "software process improvement" process. Of those organisations that have reported the assessment results the figures are presented [2]. In April 1997 over 60% of the reported organisations were assessed at level one (in '87 this percentage was over 80%). Totalling the percentages of the organisations that were assessed in one of the first three levels gets close to 100%. The naming of the successive levels are: Initial, Repeatable, Defined, Managed, and Optimising. The naming of the fourth level is the first one that suggests a certain 'maturity'.

Within the software engineering discipline there is certainly no lack of methodologies but their contribution to the maturity of the discipline seems limited. Not only the process quality is under discussion, remarks with respect to the content of methodologies can be made as well. The scientific principles, as mentioned in the characterisation of "Software Engineering", are hardly recognised in any software methodology. Strangely enough, this is quite accepted. Especially the lack of formalism is all too often regarded as an advantage instead of a handicap.

The use of formal methodologies is unfortunately the subject of extreme hyperbole or deep criticism in many 'popular press' science journals. From the claims that the authors of such articles make, it is quite clear that they have little or no feeling of what formal methodologies could contribute in industrial application. The aversion to formal methodologies is that high, that discussion exceeds the rational level and has become a religious debate a long time ago. The preconceived opinions as discussed in Hall's "Seven Myths of Formal Methods" [3] or in the article "Seven More Myths of Formal Methods" [4] are still the main belief of many. Introducing a formal method will probably not contribute to one's popularity.

In this article some important techniques and constructions, as used in the mainstream object-oriented methodologies, are examined. This article will focus on how the object modelling techniques of mainstream object-oriented methodologies are used, and the effect of these techniques on the object models that are made.

Chapter 2 contains a contemplation on the subject object-orientation. In section 2.1 the most common properties of mainstream object-oriented methodologies and the products that result from it, are summarised. Section 2.2 treats the fundamentals of the object-oriented methodologies. In section 2.3 the subject "orientation" is contrasted with
“decomposition”. Decomposition is about the model itself, the components that are used and their architecture. Orientation has much more to deal with the attitude one takes during the creation of the model. Section 2.4 provides an overview of the 'object-oriented' constructions.

Chapter 3 views on the 'practical' part of object-orientation. Along with some examples and/or cases some consequences of object-oriented modelling are considered. The fact that object-oriented is relatively new makes it a good candidate for over-enthusiasm and euphoric behaviour. Methodologies in general, and new methodologies often even more, suggest that, if followed, excellent results are guaranteed. In reality, however, every time and again one must conclude that the engineer is always responsible for the job done. A method may be of help, but never does the job.

Chapter 4 contains some observations.

2 Object-orientation

Problem analysis is the activity that encompasses learning about the problem to be solved (often through brainstorming and/or questioning), understanding the needs of the users and considering the constraints on the solution. Normally one can expect requirements ranging over a wide variety of system properties. Requirements may concern performance, interfaces, functionality, and data, the use of a specific platform or language and so on. Although it is important to fully understand all aspects of the problem, most methodologies emphasize just one aspect in their analysis model. The view taken during the first analysis steps are normally determined by the approach of the methodology that is applied. Examples of well known approaches are data-flow analysis, stepwise refinement, data-structure analysis, entity-relation diagramming (information modelling), state-transition diagramming, syntax driven program design and, of course, object-oriented analysis.

Object-orientation is one of the techniques for system modelling. The object-oriented technology originates from simulation applications. The language Simula 67 offers a number of concepts which are well suited for this purpose. Through the introduction of Smalltalk and its programming environment, the applicability of object-oriented languages were demonstrated. For some time object-orientation was seen as a technique used at implementation level; a methodology at the level of programming languages. For some it is still at that level, especially those who identify object-orientation by the programming language in use (e.g. C++, Eiffel, Ada, not Ada, Objective C, Smalltalk and not to forget Java). Others confuse object-orientation with a graphical user interface a system could have, a misunderstanding probably caused by the user interface provided in the Smalltalk environment. Nowadays, object-orientation is seen as a technique by which the system is modelled as a number of objects that interact.
2.1 Advantages of object-oriented methodologies

Unfortunately, the terminology of object-oriented methodologies is not well standardised, and there is not even agreement as to what object-oriented analysis, design or programming really is. There is no general consensus on what characterises an object-oriented method or language. But with respect to the advantages it seems that everyone agrees: “object-oriented software holds up better as requirements evolve”. And the disadvantages of the ‘old-fashioned’ methodologies are clear too: “if requirements change, a system based on decomposing functionality may require massive restructuring. Although functional decomposition seems the most direct way of implementing, the resulting system can be fragile”. An object-oriented approach will lead to a more stable system because its model is based on the underlying framework of the application domain.

In his dissertation [29] Paul Frederiks summarises the advantages of object-orientation that are encountered:

- A better integration of data and processes. In many analysis and design methodologies there is no satisfactory connection between the data models and process models. Object-oriented modelling techniques have natural integration of data and processes as a result of the encapsulation of attributes and operations in an object.
- Object-oriented systems are loosely coupled. A consequence of encapsulation of data and operations into an object is a well defined interface without side-effects.
- Reuse of classes. Objects, due the excellent cohesion and coupling properties, can be treated as stand-alone modules. Classes can be put into libraries, thus stimulating the reuse of classes.
- Cheaper maintenance. Encapsulation of data and processes makes it possible to change a class without changing other classes. Furthermore, before a class has to be changed the object library can be consulted for similar classes (generalisation/specialisation).
- Better maintenance. Conventionally developed systems are implemented to fit as good as possible for the available hardware, whereas object-oriented systems try to capture the reality as naturally as possible. The point of view in object-orientation fits better to changes that can be expected: 60% of the changes during maintenance are about changes in user requirements and data formats, only 6% are about changes in hardware [12].
- Better communication between analyst and user. Object-oriented methodologies try to describe the reality by objects which represent abstract and concrete notions of the problem area. The object model provides a better overview of the problem domain.
- Better process model. In contrast with conventional modelling techniques, models of object-oriented techniques for the different stages of the process have better connections. The products of the different stages of conventional modelling techniques often need an extra translation for their interfacing.
2.2 Fundamentals of object-orientation

Superficially the term "object-oriented" means that we organise software as a collection of discrete objects that incorporate both data and behaviour. This is in contrast to conventional programming in which data structure and behaviour are loosely connected. There is some dispute about exactly what characteristics are required by an object-oriented approach, but they generally include four aspects, or fundamentals: identity, classification, polymorphism, and inheritance [7].

In an object-oriented decomposition reality is reflected into a set of quantified discrete objects. An object is called concrete if it represents a concrete thing in reality, such as a ball, a vacuum cleaner, a light button, and so on. Objects characterising a notion are conceptual objects. Some examples of conceptual objects are the amount 3, a war strategy, or a ball. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values are equal. Objects can be compared to variables in an imperative programming language: it is quite normal to have two variables (objects) of the type (class) integer both having the value 3. Identity is in many methodologies assumed, and not made explicit.

Object-orientation is poor naming: the most interesting part of object modelling is finding the classes. Classes are groups of objects that have the same data structure (instance variables) and the same behaviour (operations). This concept, classification, in its turn is very close to that of a type. Maybe a small difference can be argued since classes offer (most of the time implicit) 'create' and 'dispose' operations whereas types leave the object life to the declaration statements and the scope-rules of the programming language.

Polymorphism, often considered as one of the basic concepts, is treated most of the time only partially in documents on object-orientation. In the article "On Understanding Types, Data Abstraction, and Polymorphism" [8], a unifying framework for polymorphic type systems is presented. Distinction is made between universal polymorphism and ad-hoc polymorphism. Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types. There are two major kinds of ad-hoc polymorphism, overloading and coercion.

In overloading, the same function identifier is used to denote different functions and the context is used to decide which function is denoted by a particular instance of the identifier. The compiler can resolve the ambiguity at compile time by giving different names to the different functions; so in a sense overloading is just a convenient syntactic abbreviation.

Coercion allows the users to omit semantically necessary type conversions. The required type conversions must be determined by the system, inserted in the program, and used by the compiler to generate required type conversion code. Coercions are essentially a form of abbreviation that improves the programs readability, but may also cause subtle and sometimes dangerous system errors. Coercion is allowed in most programming languages; automatic conversion from integer to real is perhaps a well known example.

Whereas ad-hoc polymorphic functions will only work on a finite set of different and potentially unrelated types, universally polymorphic functions will normally work on an infinite number of types (all the types having a given common structure).
In the case of universal polymorphism, one can assert with confidence that some values (i.e. polymorphic functions) have many types, whereas in ad-hoc polymorphism this is more difficult to maintain, as one may take the position that an ad-hoc polymorphic function is really a set of monomorphic functions. In terms of implementation, a universally polymorphic function will execute the same code for arguments of any admissible type, whereas an ad-hoc polymorphic function may execute different code for each type of argument.

There are two major kinds of universal polymorphism, that is, two major ways in which a value can have many types. In parametric polymorphism a polymorphic function has an implicit or explicit type parameter which determines the type of the argument for each application of that function. In inclusion polymorphism an object can be viewed as belonging to many different classes that need not to be disjoint; that is, there may be inclusion of classes. A well-known example of parametric polymorphism is the use of structures (generic types). Inclusion polymorphism is used to model subtypes and inheritance.

The fourth aspect of object-orientation, inheritance, is imbedded in the third aspect (polymorphism). The reason to mention 'inheritance' as an object-oriented fundament explicitly, may be found in the fact that promoting a more extensive use of a construction at least needs the naming of it. And with respect to this construction (inheritance) one can expect a major change in the characteristics programs will have in which inheritance is used.

Polymorphism, and especially inheritance, is often implemented by dynamic or late binding. In practice leads the extensive use of polymorphism also to an extensive use of referencing (pointers) within the implementation. Which, in its turn, stresses the need for a garbage collector. For this reason some people regard dynamic binding and the availability of a garbage collector as fundamentals of object-orientation too.

2.3 Object-orientation versus object-decomposition

Object-orientation is introduced as the new way of decomposing systems. The object-oriented methodologies claim to use quite different decomposition criteria than those applied in other (old or old-fashioned) methodologies. A closer look reveals that this is not the case. Many of the techniques used in object-oriented methodologies can be found in existing (non object-oriented) methods or seem to be just small adaptations of existing techniques. However, all object-oriented methodologies will agree on the above stated: the system is modelled as a collection of interacting objects. In this view only the type of component that is used in a system description determines its object-orientatedness, and any decomposition into object-components would deserve the label 'object-oriented'. So, along with the introduction of object-oriented methods many of the existing methods became 'object-oriented' too. All they did is some change in notation, to get an 'object' image, and added a few empirical rules to project the original decomposition onto object components. "Grouping by data" has been a good alternative. At the moment the discussion, whether such methods were object-oriented or not, got too heavy one ended in the compromise that it was 'object-oriented structured'.

In fact, the 'orientation' part of object-orientation is crucial and rarely recognised: it is the way of working, the use of dedicated decomposing criteria, the philosophical part of the method. It is easy to create a method that is functional-oriented and leads to an object-decomposition. As an example:

- take the set of functions describing the system found by a function-oriented method
- look at the parameters of these functions and for each type that occurs in the parameter lists, create an object type
- distribute the functions over the object types, based on the type of their most significant parameter.

The main challenge in transferring a method is to give insight in the way of thinking of that method. Many developers of methods must have hesitated, seeing the many faces their method got in and through the literature treating their method. This discrepancy between idea and its image is not reserved for object-oriented methodologies only, also in a book on a 'structured-analysis' one can find data-flow diagrams based on a function-oriented decomposition. The necessary data flows seemed to be added later, in stead of a decomposition as the result of an investigation of the data flows that can be recognised in the system.

Every mainstream method has a number of dialects, sometimes due to the evolution of the method, and often the consequence of the alternative books on the subject matter.

Abstraction is one of the major principles of object-orientation [5], as it is should be for every methodology. The Oxford Dictionary of Computing presents three definitions concerning abstraction:

**Abstraction:** The principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are.

**Procedural abstraction:** The principle that any operation that achieves a well-defined effect can be treated by its users as a single entity, despite the fact that the operation may actually be achieved by some sequence of lower-level operations.

**Data abstraction:** The principle of defining a data type in terms of the operations that apply to objects of the type, with the constraint that values of such objects can be modified and observed only by the use of operations.

It needs little imagination to map data-abstraction onto object-orientation, the similarities between the descriptions of both are obvious, and likewise holds the mapping of procedural abstraction onto functional-orientation.

New methods, new ideas, are normally introduced with enthusiasm, which is of course a necessary requirement to gain a field of existence. Never was the software community as enthusiastic as for object-orientation; for every activity in the software process there is an object-oriented version: object-oriented programming, object-oriented design, object-oriented analysis, object-oriented management, and object-oriented testing.

Not only every activity needs to be object-oriented, every module in the decomposition should be an object too; sometimes there is the need for a 'control' object, or a 'abstract' object, or a 'whatever' object, but every module, anywhere in the decomposition, is an object.
The same focusing can be recognised with respect to the use of inheritance, like in the early days of "structured programming" it was strictly forbidden to use the goto-statement, nowadays the use of inheritance is a must.

Under the guidance of slogans like "the integrated process approach" (every activity is ‘object-oriented’), "horizontal and vertical uniformity" (every component at any level of abstraction is an object), and "the ultimate construction for reusability" (use inheritance!!), the herd of software-lemmingineers follow the object-oriented gurus [6]. That new methods lead to enthusiasm is understandable, but sometimes this enthusiasm is resulting in overkill.

In conclusion, object-orientation should be used to find objects (classes), function-orientation to find functions, etc. The same holds for components; to capture a function, make use of a functional module, for a control abstraction a control module, etc. It is very unlikely that trying to describe a system, that reflects the real world in a natural way, will be possible with the use of one type of component. The existence of the other types of orientation, others types of abstraction, is a proof this is not the case.

In every object-oriented methodology, apart from some object model, at least a functional, dynamic or behavioural model is found. The descriptions of the techniques, that are used to create these models, contain exactly those orientations that are not object-oriented at all: they reveal the use of control and functional-orientation. Object-orientation seems to be reserved for the object model only, the other models needed to complete the system description are found by non object-oriented techniques.

In a process model the software process is made up of a number of stages such as requirements engineering, design, implementation and so on. During requirements engineering, or analysis, abstraction is used to find a model of reality in such a way that the resulting model is recognisable by the customer. The customer will be the prime reviewer of this specification and therefore, amongst other reasons, it has to be stated in the problem domain language.

In the design and implementation phase, however, abstraction is used in quite a different way. These phases are about the construction of, and finding a solution for, the system to be build in the artificial world of computing. Whereas during analysis ‘orientations’ support in capturing the requirements in a clearly structured manner; during the design and implementation the only ‘orientation’ is towards the model that has to be constructed. The nature of design and implementation is one of transforming models into design models, finally ending in code (and showing that this is correctly done). Object-orientation is reserved for the analysis phase, and moreover, it is only a part of the analysis phase: it is one of the needed ‘orientations’ for full requirements.

2.4 Constructions in object-oriented

To support object modelling, the mainstream object-oriented methodologies describe a number of investigation techniques together with a notation in which the object model is to be described. A ‘new’ type of module is provided; the object module which is normally called a class.
2.4.1 The object model

An object model captures the static structure of a system by showing the classes in that system, the associations between the classes, and the attributes and operations that characterises each class. In object-oriented methodologies the object model plays a dominant role, systems are built around objects rather than around functionality.

Classes and Objects
A Class is a set of objects exhibiting common attributes and/or functions and/or states and/or common relationships with other objects. Object that are members of a class inherit the attributes, functions, states and relationships associated with that class [9].

Brad Cox [10] is very short on describing a class: the concept of class and instance will already be familiar. In programming languages the same concept is often called type. For example, `int LeftEdge;` just says “allocate an instance called LeftEdge of class int.”.

In Object-Oriented Design [11], Grady Booch also maps the notion of a class onto the type construct but in a footnote he points out that there is a small difference in concept. Finally he ends with the conclusion that for most mortals separating the concepts of type and class is utterly confusing and adds very little value.

An object class describes a group of objects with similar properties (attributes), common behaviour (operations), common relationships to other objects, and common semantics [7].

As the diversity of the definitions above indicate, it is hard to give one meaning to the notion of class. "A class is a template for objects, describing their common behaviour" suggests that a class is at least close to the idea of an abstract data type. Based on the claims that objects hide their internal state for the outside world, one could come to the conclusion that a class and an abstract data type definition are the same (1). Brad Cox and Grady Booch seem to be in favour for this viewpoint.

Others point out that classes and types are not the same, types do not have operations for creation and deletion of objects (variables) of that type (class). Furthermore the class concept emphasizes the importance of hierarchies of classes. For some is the implicit definition of a number of operations (create, delete, the queries on instance variables) a reason to distinct between classes and types: classes need less specification texts. In these cases a precise meaning is not given, but classes are accepted to be ‘type-like’ (2).

A definition proclaiming a more concrete view can be found in "Object-oriented Software Construction" [12]. A certain hurry to start coding or the difficulty of building compilers that support separation of the definition and the implementation of a data type
justifies the idea of 'specification by representation' principle. Although the instance variables can only be addressed by invoking the methods of the class, classes are specified in terms of their implementation. This leads to the opinion that a class is an abstract data type implementation (3).

Taken Rumbaugh’s definition and considering "... describes a group of objects with similar properties ..." the 'type'-vision does not hold any longer. Besides, being a description of a type, a class is also a collection of instances, a population (4).

It is interesting to see that the definitions sometimes lead to some strange observation. An abstract class is quite commonly used, abstract classes can be found in many inheritance trees in the role of superclass. The definition of an abstract class that is found in [11] starts with: a class that has no instances. An abstract class is written with the expectation that its subclasses will add to its structure and behaviour, usually by completing the implementation of its (typically) incomplete methods. Confronting this description of an abstract class with the definition taken from Rumbaugh, we must conclude that an abstract class describes what no (none) objects have in common.

It is quite normal to support the brainstorm sessions of the analysis phase in a software development project by a notation based on diagrams. A diagram provides a quick oversight, and is therefore well suited for brainstorming. The diagrams can also be used as input for the requirements specification.

Object modelling is obviously the main modelling activity of the mainstream object-oriented methodologies. Most notations contain the same type of information. In this article an OMT/UML-like notation is used.

The basic notation for a class consists of four parts; the class-name, the attributes, the operations and the constraints. The elements that serve as input for the (formal) specification are:
(a) the class-name, uniquely identifying the class
(b) the attributes and constraints determining the state-space of the objects belonging to that class
(c) the names of the operations, sometimes with their signature, offered by the class.

Ad (b), the state space is the set of values an object can obtain. The state space, also called the valueset, consists of those elements of the cartesian product of the state spaces of its instance variables, that fulfill the constraints. Within the set of constraints two types can be recognised: attribute-constraints (Pa) a restriction on the admissible values of precisely one attribute, and a class-constraint (Pc), a further limitation based on the forbidden value combination of two or more attributes of that class.

Ad (c), the operations specify the behaviour of the class. It has become a custom to assume a number of operations to be implicitly specified: the creation and deletion of an object of that class, query operations on each of the attributes, and equality. Sometimes ordering operations (<, >, ≥, ..) are assumed available also. In the object
model the meaning of the operations is suggested by the operation name, and, if available, the signature of the operation.

In 'advanced' notations it is possible to declare class-attributes and class-operations, recognised by a prefix (i.e. "$\)$. Whereas instance variables are instantiated for each and every object of that class, a class-attribute is instantiated once for the whole class, even when the class contains zero objects. This induces the fifth meaning of class: a class is an instance of an aggregation of its class attributes together with an instance of a group (the objects of that class) (5).

**Associations**

A class is not a stand-alone entity, that is, its definition depends on more than the attributes that are declared within the class alone. The definition of a class is also dependent on the associations it has to other classes. Any dependency between two or more classes is an association. However, only those associations that describe a structural property of the application domain will appear in the object model as an association. Associations indicating a transient event, or the functionality that serves such an event, will be modelled in the dynamic model (in a state transition diagram) and as operations within a class.

Attributes can be seen as the association of precisely one object (the attribute itself) with the class it belongs to. An attribute is a part of an instance of its class. The types of these objects (attributes) are simple, normally offered by the imperative programming language that is used. At the moment there is a need for an attribute of a higher order abstraction level, the attribute could be typed by a class. This is the first reason to introduce the aggregation association. As the example shows: Heading and Section are 'higher-order attributes' of the class Document.

The existence of the classes Heading and Section have the same dependency to the class Document as attributes would have. Heading and Section are part of the class Document. A second reason to use the aggregation association is demonstrated by the class Section: whereas each Document had precisely one Heading, it can have multiple Sections. Attributes within a class normally define a single instance.

The state space of a class will be: those elements of the cartesian product of attributes and aggregations that fulfil all the constraints (the association constraints too). As the example already shows within the cartesian product one element, s, is a non-empty group (1+), described as an element of the powerset of the state space of the class Section. Due to the association constraint, O, the section numbers used in the possible values of "s" must be a closed interval in a linear ordered domain (Integer).
In the introduction of the aggregation association, some possible multiplicities were used. Within a Document the multiplicity of Heading was 'exactly one', the multiplicity of Section was 'one or more'.

Multiplicities, however, are used in all association types that denote a relation between instances (objects). In a generalisation association, a relation between classes rather than between objects, multiplicity is not meaningful.

Above the drawing conventions that will be use in this article are presented: 'exactly one', 'zero or one', 'zero or more', 'one or more', and the name of a variable of which the possible values are specified in a constraint.

Suppose that a Catalogue should be made by putting all the Headings of the Documents together. A consequence of that decision is that the existence of an object of the class Heading is no longer solely dependent on the existence of its Document. The lifetime of Heading is now depending on the Document as well as the Catalogue, it has become a self-supporting entity. [26].

Semantically there is not much difference with an association and a aggregation association. Associations have the same contribution to the state space as if they were aggregation associations. To denote the difference, Bertrand Meyer [12] introduced the notions of reference semantics and value semantics. Associations lead to references semantics, aggregation associations leave a choice: value semantics or reference semantics. Translated into implementation level terminology: an association forces to use pointers (reference) whereas an aggregation association leaves the choice between embedding the object into the object it is part of, or switch to an implementation using a reference (pointer).

Having classes, associations and aggregation associations only, the actual information is found in the attributes of the classes that are present in the model. The information
is structured by the associations and limited to meaningful values by the use of constraints. Sometimes it is more appropriate to add attributes to an association, e.g. if the information is fully dependent on an instance of the association. Such an instance of an association is called a link (compare: an object is an instance of a class).

The notation for link-attributes is presented above. The StartingDate in the example is a link-attribute, its lifecycle is fully dependent on the instance of the association 'married with'. Link-attributes are presented in a class-style manner. It may be the case, however, that the information belonging to the link is recognised as a semantical unit: leading to the introduction of an association class. (See below).

The final association type treated within this article is the generalisation association. Within object-oriented methodologies it is conspicuous to see the fast and easy shortcut that is made between generalisation-specialisation relations and the inheritance construction.

A lot of literature is found on inheritance, little though on the abstraction mechanism behind it. A further elaboration on the subject matter can be found in section 3.4.
2.4.2. The dynamic model

Those aspects of a system that are concerned with time and changes (control), are the dynamic model, in contrast with the static, or object model. Control is that aspect of a system that describes the sequences of operations that occur in response to external stimuli, without consideration what the operations do, what they operate on, or how they are implemented.

Event-trace diagrams
An event is something that happens at a point in time and, if detected by the system, a reason to act. The ‘use-case’ technique makes use of this notion to explore the systems behaviour. For each initiator of events and for any type of event the services the system should offer as a reaction on such an event, are described in a use-case. Use-cases are written in the early stages of building the requirements (they are still vague descriptions). During design the use-cases are refined in more concrete descriptions: the scenarios. A scenario is a sequence of events that occurs during one particular execution of the system.

The most detailed descriptions of the dynamic aspects of the system are found in the event-trace diagrams, showing a thread of execution. Events transmit information from one object to another, the arrow (see figure) indicates who is the sender and who is the receiver object. Objects are represented by vertical lines. In an event-trace diagram time increases from top to bottom, but the spacing is irrelevant, it shows the sequence of events. Events can happen at the same moment (Event-etc and Event-5), and may be parameterised.

State transition diagrams
A state is a stable situation of an object in which an object resides for some time [23]. A state is denoted as a rounded-rectangle. This description refers to the same notion as is used in the Mealy machine. But it must be said that this notion does not hold throughout the use of states in the diverse methodologies. Actions are assigned to states as well as assigned to state transitions.

To denote a state two representations were found: the rounded-rectangle, which is normally used, containing the name of the state and the possible activities that are performed in that state, and a bulls-eye for an final-state.
The final-state is possibly labelled to indicate the ending condition but it is not able to perform an activity. A final state implies the destruction of an object.

The initial states are indicated by an arrow starting in a solid circle, the circle may be labelled to indicate the initial condition. Switching of states is described by state transitions.

A state transition has a starting state and ending state which is possibly the same state as the starting state. State transitions are initiated by an event and during the transition an action may be performed.

Above a summary of an unstructured state transitions diagram is presented. "State-2" is the only initial state. The final state "Expired" can only be reached from "State-1". With respect to the transition "Event-1" such a transition (start state is equal to the end state) has only meaning if it is labelled with a non-empty action. A condition may be used as a guard on the transition. A guarded transition is executed when its event occurs while the guard is true.

State transition diagrams are used to model object life-cycles to get insight in the dynamic behaviour of the object. In this technique one focuses on "all the things that could happen with an object" of a class. As a spin-off one obtains a better view on the pre-condition of operations: in those states in which an operation is permitted its precondition should be true. Modelling the object life-cycle also checks the expressive power of the state space of the object: is the state space supporting in determining all the different states that should be recognised?

State transition diagrams are also used to model the dynamic behaviour of the system. This technique may be called an event driven technique. Based on the event traces found a state transition diagram is built of the system.

2.4.3 The functional model

The functional model describes computations (the operations) within a system. The functional model is the third leg of the modelling tripod, in addition to the object model and the dynamic model. The functional model specifies what happens, the dynamic model when it happens and the object model what it happens to.
The functional model shows which values depend on which other values and the functions that relate them. The diagramming technique that is used to model functionality is mainly data flow diagramming. This modelling technique, however, is only used in a limited set of all object-oriented methodologies.

Some methodologies proclaim that the needed operations are found during object modelling and dynamic modelling already. The operations typically belonging to a class are modelled during object modelling, some of those even implicitly. Modelling the object life-cycles give rise to the definition of boolean operations that support robustness, and the set of operations, which are perhaps more dedicated to the application at this time, can be extended. The set of operations that completes the needed functionality of the system are found in the activities that are performed in the states of the system’s dynamic model and in the actions defined at the state transitions. Data flow diagramming is seen as the dominant model in the ‘structured’ methodologies. More on data flow diagramming can be found in [7,9,27,28].

3 Now practical, O.O.

3.1 Mainstream methods and techniques

In the past twenty years, a large number of methodologies have been introduced. Up to the late eighties most methodologies were "structured", a nearby synonym for "data-flow oriented". In the past ten years, however, at least 20 object-oriented methods were proposed in book form and many more in papers. In this article the 'mainstream' object-oriented methodologies are subject of discussion. To provide an idea, which and what type of methodologies are meant to be ‘mainstream’ a short overview is given.

Shlaer and Mellor partition systems into domains, where a domain is defined as a part of the world with its own conceptual space of rules and behaviour. Example domains are the implementation domain (programming languages, operating systems, etc.), the service domain (containing utility functions for user interfaces, mathematical libraries, etc.), and the problem domain (also called: application area, application domain, subject domain or problem space). Large domains will be partitioned into sub-systems that are loosely coupled but have close cohesion. This partitioning is an explicit step in for instance the Octopus method. Each subsystem is modelled as a collection of communicating objects. Shlaer and Mellor wrote "Modelling the word in data"[14] in 1988. It contains an object-oriented variant of information modelling. In 1992 "Modelling the world in states"[15] was written. It describes a pragmatic approach to model the dynamic behaviour of a system in terms of state and process models.

In 1990 Coad and Yourdon [5] published their book on object-oriented analysis. The conceptual decomposition of the system is described in an object model, consisting of the relevant classes. The structure layer adds the containment relations
and inheritance trees. In the subject layer one can recognise a 'role' investigation, one of the investigations defined in OMT. The object model is refined with attributes, after which the functionality is explored in a service layer. Services are described in natural language, but the idea of state transition diagrams scenarios and event traces can be recognised in the specifications as a source of inspiration.

Booch [16] introduced his method for software design in 1986 with “software engineering in Ada”. This work has been one of the inspirations of many object-oriented methodologies. In 1991 Booch published “Object Oriented Design” [17], and a successor (second edition) in 1994. Booch represents the structure of a software system by means of a class diagram and the behaviour of the objects by means of state diagrams. The rather simple state diagrams introduced in 1991, are replaced by a statechart-like notation in 1994. As were communications represented by timing diagrams in 1991, in 1994 these timing diagrams are sequence diagrams turned on their sides.

The Object Modelling Technique (OMT) was introduced by Loomis et al [18] and popularised by Rumbaugh et al [7]. A significant update, OMT95, was published in 1995. The decomposition of the system into objects is represented by the object model, again an object-oriented variant of information modelling. The behaviour of objects is described in the dynamic model; the notation is based on a statechart variant. Object operations are defined in the functional model. In OMT95 use cases (adopted from Objectory) are used to complement the problem statement. In the OMT-dialects that exist nowadays one can encounter techniques as: Data-Flow diagrams, scenarios, event traces, event-flow diagrams, state-transition diagrams and object life-cycles to be of support during the analysis and design phase.

Behaviour-based methodologies have been developed as alternatives to the methodologies based on conceptual models, e.g. information modelling. The underlying idea behind these methodologies is simple: communication aspects are analysed first, because in the final stage the system will be composed of a collection of communicating objects. Among the behaviour-based methodologies, Class-Responsibility-Collaboration cards, introduced by Wirfs-Brock [19,30], is well known. Objectory is the commercialised version of Object-Oriented Software Engineering [13]. The external functionality is explored in a use case driven approach, the preparation of the conceptual decomposition which in its turn is represented by a domain object model. Objectory can also be seen as one of the major behaviour-based methods.

Fusion, also an evolving method, defines a decomposition of a system by first specifying a domain model, which is again a variant of an information model, of the application area. The system decomposition is found after that by outlining the systems boundary in the domain model. Everything inside the boundary is seen as part of the system, everything outside the boundary is assumed to be part of the environment.

CODARTS [20] has its roots in structured analysis and design. Recently, it has been complemented by a domain model which adds some more object-oriented flavour. The domain model defines several viewpoints for the analysis of the system. In one of these views, the data flow diagram notation is used to develop object
communication diagrams. This shows the flow and control between nodes that are called concurrent objects.

In "Object-Oriented Technology" Awad et al. [21] describe their method as follows: the OMT and Fusion methods are the basis for the development of Octopus. The object model notation of OMT enables compact expression of all the necessary details, and the separation of structural, functional and dynamic aspects makes the models easier to build and understand. Basic separation between the analysis phase, concentrating on describing the external behaviour, and the design phase, concentrating on the internal behaviour of the application is borrowed from the Fusion method. The Octopus method is extended with techniques to cope with the characteristic problems of software development for real-time embedded systems.

The Unified Modelling Language (UML) [22] arose as a joint effort of Booch, Jacobson and Rumbaugh to unify the existing notations for object-oriented software specification. It combines the notations used in the ‘Booch” method, Objectory and OMT, applies some simplifications, and extends this with new features in diagrams. However, the basic structure of the notations used in these methods remain recognisable. The intention is that it will be used as a diagram convention in those methods and probably in other methods as well. UML can therefore not be seen as a method but only as a notation, as the name suggests.

Reviewing the mainstream object-oriented methodologies one can argue that the differences are cosmetic. And with respect to the notations that is certainly true: in every methodology one can find syntactical support to express any type of abstraction and any type of component. Differences at notational level are mainly found in the shapes they use.

Already in the ‘structured’ decade three models were recognised: the functional model, the control model and the data model. In ‘structured’ methodologies normally the functional model acted as the main model, in ‘real-time’ versions heavily supported by the control model. In hardly any ‘structured’ methodology the data model was taken into account.

Object-oriented methodologies still recognise these three models, however, slightly modified. The main model of object-oriented methodologies is the object model, an enhanced data model. The elements of the object model are classes in which data and operations are integrated. The dynamic model has become the new name of the control model. The functional model is not renamed. Object-oriented methodologies perform their analysis with an ‘object-vision’ (object-orientation) thus leading to an object model (object decomposition). In that distinguish object-oriented methodologies themselves amongst other methodologies.

To determine the feasibility of a software project, a project will start with the construction of a draft version of the problem statement. Usually, such draft version is made through brainstorming and/or questioning. This is a starting point for every methodology. Within the mainstream object-oriented methodologies two strategies can be recognised. Some start to refine the problem statement first, for example through further investigation using use-cases. Others start object modelling right away using the problem statement and likely supported by a domain expert.
Further investigation on the problem statement has often a functional nature, and as such it is endangering the ‘object-vision’ needed to be object-oriented. The disadvantage of not doing further investigation on the problem, is that the draft problem statement is too vague and lacking important aspects of the problem. As a compromise some methodologies propose to complete the problem statement first, to get a better view on the problem, but neglect the ‘functional’ supplement during the construction of the object model. The functional nature of the supplement would diminish the object-orientation. Nevertheless, the ‘functional’ supplement can be used to validate the expressive power of the object model. Nowadays, problem investigation with use-cases is regarded as the most popular technique to complement the problem statement.

To support the recognition of objects (in fact classes) three techniques seem to be in favour: modelling ‘nouns and verbs’, responsibility driven modelling and information modelling. The ‘noun and verbs’ technique is simple: given a text, normally the problem statement, nouns are seen class indicators, the verbs are candidates for operations to be defined. Section 3.2 will treat the ‘nouns and verbs’ technique in more detail. Of the responsibility-driven technique CRC-charting [30] (Classes, Responsibilities and Collaborations) should be mentioned. Information modelling is probably the technique that most extensively used. That is, most object modelling techniques are derivatives from information modelling. Information modelling has its roots in data base design, data structures (attributes, records, tables, and the data base itself) were analysed using the technique of entity relationship diagramming. Focusing, therefore, is on the data of the system, or as expressed by Rumbaugh: the object model specifies what it happens to.

Methodologies, in general, start their analysis at an informal and vague level and tend to add detail in the successive phases (The program code is normally the first document that is completely formal). “From vague to concrete” would appropriately describe the nature of most development cycles. Usually, object models are represented in diagram form, leaving lots of aspects unspecified. (state space, pre & post conditions of operation, etc.) During design, however, one can recognise techniques to correct the previous mentioned omissions. Object life-cycle modelling is a broadly accepted technique within the construction of the dynamic model. Representing the object’s life in a state transition diagram gives insight in; the operations needed (they are found as activities in the states or as actions on the state transitions) and on the pre and post conditions as well. Investigating in which states an operation is available, and in which states it is unauthorised, gives insight in the pre and post-condition of the operation. Furthermore is the state space checked against its expressive power.

Another part of analysis, contained in the dynamic model is the control of the system. Normally the use-cases, or the functional descriptions that where made in order to complete the problem statement, are refined into scenarios. In a next step the scenarios are translated into event-traces, the lowest level of control specifications. In these steps new operations are encountered, operations specifically needed in the application. These application dependent operations are added to the object model. The overall control of the system is reflected in state transition diagrams.
3.2 Finding the classes

One of the main difficulties in object-orientation is finding the relevant classes and objects. One of the most popular ways of finding objects, classes and their methods is by giving short problem statements and/or use cases, sometimes also called scenarios, and investigating the nouns and verbs used. In the literature one can find many discussions on the "nouns and verbs" technique, some claiming that it would lead to too many classes and, if applied to use-cases, to a functional decomposition.

In his book "Object-Oriented Software Construction" [12] Bertrand Meyer gives a warning with respect to this way of finding classes. He classifies it as a 'simple-minded' technique, which can give only rough results. Whereas Bertrand Meyer gives this technique the disclaimer "finding far too many candidate classes", others [23] explicitly demand an unrestricted selection of classes. Within the set of candidate classes further selection is carried out, based on findings such as:
- Does a class have a clear responsibility?,
- Does it have a self-supporting role in the problem domain?
- Or, is explicit information needed about this class, or can it serve me?
The 'real' classes and objects will be those for which the above questions are answered with a clear "yes".

With respect to the appliance of this analysis technique to use cases, one tends to be sceptical: it would lead too much to the same functional decomposition one would have found with the old-fashioned technique. The explanation for this is the functional-oriented way of describing a system when using use cases. An entry point for more detailed discussion is to be found in the article: "Be careful with Use Cases" [24].

But there is even more to it: the texts used during this investigation are biased by the problem on hand, thereby influencing the model that follows from it.

Object-oriented decompositions are said to be the natural way of specifying systems: the real world is reflected in an abstraction in a very recognisable way. Besides the remarks above, there is more to be said on the analysis technique based on the investigation of nouns and verbs. Even the careful application of such an investigation will lead to a decomposition which is highly dependent on the view that the writer of the problem statement or use case had. To illustrate the idea, three versions of the same (small) system are presented and models that may result from these texts are described.

The intercom system (1)

In order to improve communication in an office building, an intercom system is installed. With such a system it is possible to request a connection from one office to another. Each office has a room number which will be used in the request for a connection. However, requesting a connection is only confirmed under the following conditions;
- Both offices are free (are not involved in any connection)
- Both offices are in normal mode
- No 'self-requesting' is done
To avoid unwanted disturbance, an office can be shielded from possible connections by putting itself in private mode.
Following this problem statement we could easily conclude that an Intercom system consists of a group of Connections. A Connection relates a calling Office with a called one. Each Office is identified by its room number (qualified attribute) and has a mode (normal, private). \textit{Request}, \textit{EndOfRequest} and \textit{SetMode} are methods (or operations) that can be expected to be defined and their behaviour is probably described in more detail in a use-case for each one of them.

\textbf{The intercom system (2)}

In order to improve communication in an office building, an intercom system is installed. Such an intercom system implements the requests for communication made by people in the building. One can make a request by giving the number of the office one would like to connect to. A request is granted in case:

- neither one of the offices is connected in a pending request
- neither one of the offices is in private mode
- no attempt is made for connecting to oneself

To avoid unwanted disturbance, an office can be placed in private mode, so it can not be connected to.

In this specification the usage of the most important noun and verb are interchanged, the class \textit{Connection} becoming the class \textit{Request}. But still the same Intercom system is described. Now an \textit{Intercom system} has become a set of \textit{Requests}, and the \textit{Request} is the object that characterises the link between two \textit{Offices}, in the same way as \textit{Connection} did in the previous example. Operations (or methods) that can be expected are: \textit{Connect}, \textit{DisConnect} and \textit{SetMode}.

Perhaps in this case, due to the statement that a request may be pending, it will get an attribute called status with two possible values (pending, disconnected). In that case the Intercom system is characterised by the history of requests.

The only commonality left is the structure we found: The Intercom system is a set of tuples (pairs) in which a tuple characterises the link between two \textit{Offices}. In our first example the tuple was called a Connection, in the second one request.

If we take in mind the statement “the natural decomposition”, the first questions will arise. It is not that clear that a \textit{Request} and a \textit{Connection} are the same, nevertheless they are both describing the same notion, the link between two \textit{Offices}. Things become even worse, however, if a more radical approach is taken than just changing the nouns and verbs. To illustrate this idea a third problem statement:
The intercom system (3)

In order to improve communication in an office building, an intercom system is installed. Therefore, each office gets an intercom. An intercom is addressed by its unique number, which is of course the same as the number of the office in which it is installed. An intercom has two modes: normal and private. An intercom is set to private if no disturbance through calls is allowed. Another intercom can only be contacted if it is free and not in private mode.

This problem statement will lead to a totally different naming and structure as well. The Intercom system will become a set of Intercoms, identified by a number, being in a mode and may be contacting another Intercom.

Given the high level of non-determinism, as the previous examples showed, it still occurs to be the "natural" decomposition? Or is object-oriented analysis not so easy, natural or straightforward as it is said to be?

Viewing the examples the correspondence between the Intercom system and a graph structure is evident. Examples 1 and 2 lay emphasis on the edges of the graph whereas in example 3 the nodes play the central role. Such imbalances in models are often not recognised, which is understandable if we take the focusing on the problem into account. As long as the models supports the system description, there is hardly any need to correct the model.

3.3 Application driven modelling

One of the conspicuous properties of all methodologies is their starting point for analysis: they use some description of the problem itself. Whether the problem is defined in a problem statement, a set of use-cases, a user-requirements specification or a functional specification, they all are directly inspired by the problem. After choosing the problem as a source of inspiration it is likely one is invited to focus on the problem and the resulting model will become problem dependent: a setback for the claim that it is reusable.

An attempt to construct less problem-dependent models is made by those who try to introduce 'domain knowledge' as a source of inspiration. The domain knowledge is not problem-specific but can be used for the class of problems belonging to that domain, just like mathematical science is not dedicated to one specific mathematical problem but to a large collection of mathematical problems. In order to deal with such problems, a mathematician uses his own language, his own abstractions of reality and his own logic.

Finding the reusable and natural model becomes a question of analysing and defining the language of the problem domain, not the problem itself. Given such a
‘language’ model, all ingredients needed for an arbitrary problem statement are available.

In the previous section, finding the classes, the possible effects of the ‘nouns and verbs’ strategy on the structure of, and the names found in the model were discussed. In this section a further elaboration is presented on some properties of the model that can be expected, if the problem statement is chosen as a start for requirements engineering.

In a graphical user interface, a number of windows are displayed on screen. The class Window will be one of the classes to be recognised. Typical instance variables will be: Length, Height, and its position. Likewise, typical operations will be: Display, Move, Resize and so on. For those who are familiar with the literature on object-orientation the presented model should be recognisable.

This model of the class Window works fine until a change request introduces the idea of work spaces, or virtual screens. Windows are displayed on one or more work spaces, on possibly different positions. One of the Work spaces is displayed on the physical screen.

To be able to support this new functionality, the previous model has to be revised. The window position is no longer a characterisation of the Window itself. A better model would be one in which the Window position has become a association-class.

The operation Move is moved to another class and the operation Display is at least redefined to one with a parameter (the position) or defined elsewhere as well. The question, whether the first model of the class Window was good, bad or ugly, will rise. Answering the question is not easy; the model was sufficient at the time it was made. The reason the first model was chosen is obvious; it was made within one context, the problem on hand, and within that context the notion of Work spaces was not recognised.

Closely related to the methodologies of object-oriented analysis are those of domain analysis. Whereas object-oriented analysis typically focuses upon one specific problem at a time, domain analysis seeks to identify the classes and objects that are
common to all applications within a given domain. If you are in the midst of a design and stuck for ideas as to the key abstractions that exist, domain analysis can help by pointing you to the key abstractions that have proven useful in other related systems. Domain analysis works well because, except for special situations, there are very few truly unique kinds of software systems [11].

The conclusion that is left to the reader is that object models as a result of domain analysis are more reusable than problem-oriented object models. Domain models are generally applicable. In a reverse direction one could state that, the more problem specific an object model gets, the less reusable it will be.

Object-oriented methodologies use the problem statement, or the set of use-cases describing the application, as the source of inspiration, thus constructing a quite problem specific model. And, moreover, in many of these methodologies the object model found during analysis is adjusted (down-sized) to the application that has to be build. So it ends up being even more specific.

Suppose, that the class Date, a commonly used class, is taken from the class library and the interface looks like:

<table>
<thead>
<tr>
<th>Date</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Day#</td>
<td></td>
</tr>
<tr>
<td>Integer Month#</td>
<td></td>
</tr>
<tr>
<td>Integer Year#</td>
<td></td>
</tr>
<tr>
<td>Date CurrentDate ()</td>
<td></td>
</tr>
<tr>
<td>Boolean Today (Date checkdate)</td>
<td></td>
</tr>
<tr>
<td>Date Tomorrow ()</td>
<td></td>
</tr>
<tr>
<td>Boolean Future (Date checkdate)</td>
<td></td>
</tr>
<tr>
<td>Date Yesterday ()</td>
<td></td>
</tr>
<tr>
<td>Boolean Past (Date checkdate)</td>
<td></td>
</tr>
<tr>
<td>Integer Daydifference (Date first,last)</td>
<td></td>
</tr>
<tr>
<td>Boolean LeapYear (Date checkdate)</td>
<td></td>
</tr>
<tr>
<td>Date NextLeapYear ()</td>
<td></td>
</tr>
<tr>
<td>Integer Age (Date birthdate)</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

After such an impressive interface it is imaginable that one decides not to use this class. The operations may have been, and probably still are, very useful in an application, but there are too many operations available that are never needed in ‘normal’ applications. Adding operations specific for one application to the interface of the class makes the class less reusable. In the majority of object-oriented methodologies, however, it is normal to define the application dependent operations within the class definition.
3.4 Modelling inheritance trees

The inheritance principle forms the basis for a powerful technique of explicit expression of commonality. Inheritance allows us to specify common attributes and operations once, as well as specialise and extend those attributes and operations into specific classes. Object-oriented analysis uses inheritance to explicitly express commonality, beginning with the early activities of requirements analysis [5].

The ability to factor out common properties of several classes into a common superclass and to inherit the properties from the superclass can greatly reduce repetition within designs and programs and is one of the main advantages of an object-oriented system [7].

A class is a descendant of one or more other ones if it is designed as an extension or specialisation of these classes. This is the powerful notion of (multiple) inheritance [12].

Generalisation, specialisation, inheritance trees, without these no object-orientation is possible. An object model without some inheritance tree is at least suspicious. In this section is discussed whether inheritance fulfils the promises that are made. Are inheritance trees the ultimate solution to reusability? Is reality truthfully reflected in the model and thereby the 'natural' decomposition?

A requirements specification of a small case is modelled using an inheritance tree. Within this case, however, are all the elements that play a role in the discussion are present. It must be said that these elements seem of minor importance but that is due to the 'condensation' of the case itself. So will the sharing of attributes and operations of two types stand for a number of significant commonalities of these types, and the differences stand for significant differences.

In this case the class Person plays the central role. Each person is either a man or a woman. The classes Person, Woman and Man are needed in an application where simulation is done of the behaviour of waiting queues. Given a number of counters, a queue of persons is waiting for each counter. A counter will served the persons from its queue one by one. Depending on the gender of the person, male or female, service is adjusted. In this case is the required functionality as follows:

- Given a person, a man or a woman: the physical age.
- Given a person, a woman or a man: her or his name.
- Given a person: whether it is a man or a woman.
- Given a woman, or a person that is a woman: the number of children.
- Given a man, or a person that is a man: his secret.
- Given a person, a woman or a man: whether the marriageable age is reached. This function differs for a man and a woman, because the marriageable age of a man is 18 and the marriageable age of a woman is 16.

Within this application domain each man is characterised by his name, birthdate and his secret. Each woman is characterised by her name, birthdate and the number of children. The types String, Boolean, Date and Integer are assumed to be pre-defined types.
In object-orientation, the commonalities are located in supertypes (the generalisations) and the subtypes or descendants describe the specialisations. This is done by adding information to the already inherited information of its supertype. The following model is proposed:

Locating the common instance variables in the superclass Person is no problem. To allocate the operations is less straightforward, some operations can not be defined within the class itself. For those cases, the solution within object-orientation is found in the declaration of "deferred" or "virtual" operations. The definition is postponed to the subclasses.

In this discussion an operation may have the keyword “specified”, as a shorthand for the specification of the operation itself. The operations are that simple, that it should not lead to any inconvenience. Furthermore, operations that are direct queries on instance variables and operations dealing with the object life-cycle (Create, Delete) are implicitly defined, as it is a custom in mainstream object-oriented methodologies. In case an operation can not be specified in the superclass nor in all of its subclasses a "partial deferred" is accepted, provided that the condition for which classes the operation holds is testable.
The model together with its operations becomes:

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>String Name</td>
</tr>
<tr>
<td>Date Birthdate</td>
</tr>
<tr>
<td>[ State space: Name x Birthdate]</td>
</tr>
<tr>
<td>Integer Age (Date currentdate) specified</td>
</tr>
<tr>
<td>Boolean IsMan deferred</td>
</tr>
<tr>
<td>Boolean IsWoman deferred</td>
</tr>
<tr>
<td>Integer #Children partial deferred(IsWoman)</td>
</tr>
<tr>
<td>String Secret partial deferred(IsMan)</td>
</tr>
<tr>
<td>Boolean Marriageable deferred</td>
</tr>
</tbody>
</table>

[ State space: Name x Birthdate x #Children ]

[ Woman ]

| Integer #Children |
| [ State space: Name x Birthdate x #Children ] |
| Boolean IsMan specified |
| Boolean IsWoman specified |
| Boolean Marriageable specified |

[ State space: Name x Birthdate x #Children ]

[ Man ]

| String Secret |
| [ State space: Name x Birthdate x Secret ] |
| Boolean IsMan specified |
| Boolean IsWoman specified |
| Boolean Marriageable specified |

And this is the typical model we could expect, it follows the 'object-oriented methods' nicely. Looking at this model a number of drawbacks can be observed.

People regard their environment in terms of objects. Therefore it is simple to think in the same way when it comes to designing a model. A model which is designed using an object-oriented technology is often easy to understand, as it can be directly related to reality. Thus, only a small semantic gap will exist between reality and the model. The abstractions made from reality should be those that eliminate uninteresting details only. The model should comply with reality, not only containing the needed information but also reflecting the structure of reality truthfully.

Models that are not fulfilling the need for information will easily be classified as being incorrect. Nevertheless, models can still significantly deviate in structure and naming: The resulting model is depending on the orientation (functional, data, ...) the analyst has used, or the structure and naming of the model may be imposed by a view on a possible implementation. Deviations, due to the different 'orientations' will be inspected with the notion of compliance. Implementation bias will be recognised as "overspecification", one of the seven sins of the specifier. [25]. These specification sins will be the second metric to judge a model, followed by a short elaboration of its cohesion and coupling characteristics.
As a discussion aid, the state space of a class was introduced in the section on constructions in object-oriented. With the use of the state space, the compliance of the model is checked: is every man person? (and likewise for woman) and is every person either a man or a woman? These propositions hold in reality but not in the model. The so-called 'population' view is not reflected in the model.

The functions *Marriageable*, *IsWoman* and *IsMan* are specified in the class *Woman* as well as in the class *Man*. This has been done in order to fulfil the deferred declaration made in the class *Person*. Besides, that the postponement of function definitions is in fact forward referencing (one of the seven sins of the specifier [25]), it also creates a strong coupling between the classes involved. This strong coupling is well demonstrated by the, for the classes *Man* and *Woman* unnatural, specifications of the operations *IsMan* and *IsWoman*. This type of coupling, introduced by the use of deferred operations, is typical in specifications with commonalities located in a superclass. The commonalities, simply and solely, are not sufficient to specify all operations, which will in its turn unavoidably lead to the use of the deferred clause.

In the next model the previously mentioned disadvantages are solved. The class *Person* consists of either a *Man* or a *Woman*. The instance variable *Gender* will have the value True in case the *Person* is a *Woman*, False otherwise. The *Gender* will also be used as a discriminator in the constraint on *Person*.

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean <em>Gender</em></td>
</tr>
<tr>
<td>[ State space: { (True,w)</td>
</tr>
<tr>
<td>Integer <em>Age</em> (Date <em>currentdate</em>) specified</td>
</tr>
<tr>
<td>Boolean <em>IsMan</em> specified</td>
</tr>
<tr>
<td>Boolean <em>IsWoman</em> specified</td>
</tr>
<tr>
<td>Integer #Children specified</td>
</tr>
<tr>
<td>String <em>Secret</em> specified</td>
</tr>
<tr>
<td>Boolean <em>Marriageable</em> specified</td>
</tr>
</tbody>
</table>

\[(Gender \Rightarrow (a = 1 \land b = 0)) \land \]
\[(\neg Gender \Rightarrow (a = 0 \land b = 1))\]

<table>
<thead>
<tr>
<th>Woman</th>
</tr>
</thead>
<tbody>
<tr>
<td>String <em>Name</em></td>
</tr>
<tr>
<td>Date <em>Birthdate</em></td>
</tr>
<tr>
<td>Integer #Children</td>
</tr>
<tr>
<td>[ State space: Name × Birthdate × #Children ]</td>
</tr>
<tr>
<td>Integer <em>Age</em> (Date <em>currentdate</em>) specified</td>
</tr>
<tr>
<td>Boolean <em>Marriageable</em> specified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Man</th>
</tr>
</thead>
<tbody>
<tr>
<td>String <em>Name</em></td>
</tr>
<tr>
<td>Date <em>Birthdate</em></td>
</tr>
<tr>
<td>String <em>Secret</em></td>
</tr>
<tr>
<td>[ State space: Name × Birthdate × Secret ]</td>
</tr>
<tr>
<td>Integer <em>Age</em> (Date <em>currentdate</em>) specified</td>
</tr>
<tr>
<td>Boolean <em>Marriageable</em> specified</td>
</tr>
</tbody>
</table>
To show the relation between Person, Man and Woman an instance, I, of the class Person, and according to the definition of the state space $I = (G,P)$, should be interpreted as $P$. In this model, the ‘compliance’ rule is not violated. Every element of Man, and every element of Woman is element of Person. No deferred operations are declared so neither unwanted forward referencing nor unwanted coupling.

But, from the point of view of the implementor, the first model has some qualities that can be used advantageously during implementation. The specifications of the classes Man and Woman have clearly much in common. Localising these commonalities will support the efficiency of the solution.

It is time to recall the software process models in which a clear distinction is made between analysis and design and their goals. It seems that the practical situation often blurs these phases. The first signs of ‘rush to code’ can be recognised in the need for ‘efficient’ specifications (which is in practice always a synonym for overspecification).

The second model is, because of the better fit to the customer, preferred to be placed in the requirements specification, leaving the idea of the inheritance tree as the result of a first design step:
In this model the contradictory views on a superclass, being a union of populations and only containing the commonality of these populations, are separated. The class Person contains the ‘union’ view, whereas the commonalities are located in the abstract class WomanManCom. This class is introduced for implementation purposes during design as an aid for an efficient solution. In this final model all advantages, without having the disadvantages, are realised.

3.5 Specification by representation

Separation of concerns of specification (definition) and implementation is a very important issue in software engineering, on all levels. During system specification one concentrates on the out-side of the system, not on the internal details (implementation) of the system. Furthermore, during implementation, considerations about efficiency are important. Implementation details in a specification are often disrupting the clarity of that specification. In this section will be shown that in object-oriented methodologies the separation of concern is often neglected.

In the example of the previous section the class Woman was diagrammed. Within that class the instance variable “Integer #Children” was declared. In fact the attribute constraint expressing that one can not have a negative number of children should be stated. Attributes are not recognised as classes. So at every point where the #Children is needed, all aspects belonging to #Children have to be specified again. It is very unlikely that the #Children can be negative for any species on our world. In fact, the constraint C, is a class-constraint in a class #Children and has nothing to do with the class Woman. Besides that “Integer” and “#Children” differ in valueset, their behaviour is different too. It is quite normal to multiply integers resulting in integers, but multiplying #Children with #Children makes no sense (it would result in square #Children???). #Children and Integer can not be interchanged, Integer is a possible implementation for the class #Children. The function that projects the implementation values, integers, onto the values of #Children is a partial function with all the non-negative integers in its domain and the projection of the elements is their identity.

In retrospect, the conclusion must be made that all the leaves in an object model, classes that can be specified with a known (program language) type as universe, are not recognised as class at all. So typically, Length, Height, Salary, Room numbers, Age, etc., are never recognised as a semantic unit. A small semantic gap.
At the level of classes, which are specified as abstract data type implementations, the semantic gap becomes bigger. Consider the classes Rectangle and Square, both classes in a large inheritance tree specifying all kinds of quadrangles. In order to be memory-space efficient the attributes of a Square are position and Width. The shape of a Rectangle is less straightforward, for the size of the edges may differ. Now is the question, who is inheriting who? Based on the consideration: a specialisation inherits from its generalisation, the Square should inherit from the Rectangle. A Square is always a Rectangle but not vice versa. Therefore Square is a subclass of the super class Rectangle. The instance variables suggest, however, that the class Rectangle should inherit from the class Square. All attributes are inherited and one attribute, Height, has to be added. The confusion is caused by the difference in the chosen implementations and the abstractions they described. The latter, the abstractions, are not described at all but are the source of the ‘generalisation-specialisation’ consideration. If Square and Rectangle were specified in the same universe, the ‘generalisation-specialisation’ could be recognised. Suppose, as a universe two diagonally opposed positions were chosen, a universe suitable for both shapes. The comparison between the two classes becomes a lot easier. The constraint stated in the class Square is stronger than the constraint of the class Rectangle, the weakest constraint possible (True) and normally omitted from the diagram.

Initially, the definitions of Square and Rectangle were inspired by a space efficient choice of attributes, and thereby each class had its own universe. The chosen implementation confused the designers of inheritance trees. In the second specification, however, the same universe was chosen but it violated the efficiency of the implementation. Another cause of discussion on inheritance trees is the relation between generic types. Give some thoughts on the structures Set(type t) and Bag(type t). Which class should inherit from which class? Is the class Bag(type t), having an extra characteristic, namely, an operation to determine the frequency of an element, a specialisation of the Set? Or is the class Set(type t) a specialisation of the class Bag(type t), because as stated before, every element of Set(type t) is also an element
of Bag(type t)? The Set(type t) would have a frequency of one for all its elements if seen as a Bag(type t).

4 Observations

The mainstream object-oriented methodologies reveal similarities with the ironically described ‘life-cycle of a methodology’:

- A methodology starts in a project that is successful and an alternative procedure is used to reached the project goals.
- A buzzword is introduced to denote the alternative procedure.
- An impressive methodology, recognised by the buzzword in its name, is designed. Probably, it has some links to the original idea behind the alternative procedure.
- Many seminars, courses and lectures are given on the methodology, meanwhile evolving.
- The methodology is described in books: an ‘original’ version, enhanced versions as a result of method evolution, ‘super’ versions as a result of unification with methodologies having the same buzzword in their name, a ‘practical’ version, a ‘real-time’ version and so on.
- Lots of projects later, the software engineering community is aware that projects still fail, and is seeking for the next generation of methodologies that, this time, really will solve their problems. The buzzwords are already waiting.

It will be clear that the requirements for the next methodology are:
(a) Informal, for reasons as discussed in “...Myths on Formal Methods” [3,4]
(b) No documentation other than code, for documentation slows down the development process. And, secondly, maintenance is done on code and not on documentation, so documentation is useless after the system is implemented.
(c) The result of the methodology has excellent properties, the product will satisfy the eleven quality factors as described in RELiability and QUality of European Software Technology [31] fully.
(d) Everyone can use the methodology.
(e) The methodology is learned (at level of application) within a period of two weeks.
(f) The methodology can be applied in every type of software system.

Many experts believe that the object-oriented methodologies are a step forward in solving the software crisis, even though they lack:

(1) a formal foundation, (in general the syntax of graphical representations is explained by an enormous number of pictures, whereas the semantics is not defined at all)

(2) common terminology (e.g. five different meanings of a class were found)

(3) a common view on what object-orientation is (using a object-oriented language, graphical user interfaces, or a decomposition into interacting objects).

Within the world of ‘object-oriented’ there are methodologies found that, though these methodologies result in an object decomposition, can not be seen as ‘object-oriented’ cause they lack the ‘object-vision’. Furthermore, within the class of object-
oriented methodologies some extreme implementations can be found: trying to
describe everything as an object, over-emphasizing the use of inheritance, declare
every activity to be object-oriented.

Within industrial environments a certain ‘rush to code’ can be observed and
methodologies seem to adapt themselves to that:

• specification by representation
The first example to mention in this context are the attributes, directly declared as an
instance of a type that is offered by the programming language. In section 3.6 is
shown that attributes are just classes.

It needs no explanation that the class definitions are the second example, it is
found in one of the meanings of the class: a class is an abstract data type
implementation.

• implicit definitions
The state space of a class is normally not specified. The reader must get a feeling for
the valid values of an object, based on the attributes and associations of that class and
of course the constraints.

Within methodologies it becomes a custom not to specify a set of ‘standard’
operations but, they are assumed to be available. Whether this is convenient or not
depends on the ‘predictability’ of those operations. Are all equality operations over
all classes the same so they do not need to be specified?

For example: consider a class, Fraction, having two instance variables named
nominator and denominator both of type integer. What is the state space? Is there a
constraint needed? How are the ‘standard’ operations defined?

Leaving out specifications that seem to be trivial, are especially endangering the
correct functioning of the system, the moment there is an anomaly.

• incomplete specifications
Special warnings are found in the prescriptions that methodologies offer with respect
to the level of detail. “At most 50 uses-cases”, “Seven, plus or minus two”, “No user
interface aspects” are examples of heuristics that lead to incomplete specifications.
Indeed, all too often the user interface is not specified at all (such a detail, often more
than 50% of the total code, is left for the designer and implementor of the system).
Also the definitions that are made are refined during design and implementation,
which means in practice, that significant details have to be added. Already in section
3.1 the conclusion was made that ‘from vague to concrete’ characterises the
development cycle appropriately.

• overspecification
All examples of “specification by representation” are in fact examples of
overspecification. Moreover, it is shown in section 3.4 that the early introduction of
inheritance trees leads to overspecification. Another example of overspecification can
be recognised in the algorithmically specified operations.

In section 2.1 the advantages of object-oriented methodologies were summarised.
Some remarks upon these advantages could be made.
Natural integration and better communication
In section 3.2 is shown that the object models easily differ in their class-names and structure.
In section 3.4 the inheritance tree introduced a model that did not comply to reality.
Not specifying attributes as classes limits the vocabulary of the object model to a subset of the vocabulary that is used in the problem domain.
Due to ‘specification by representation’ the vocabulary is expressed in a possible implementation, in contrast to the abstraction it encompasses.
Omitting relevant details during analysis makes the specifications vague.

Well defined interface
Interfaces contain only a subset of the operations (there is a set implicitly available) and operations are not formally specified.
The state space is usually not defined.
Classes are coupled through the inheritance tree(s) they are part.

Reuse
In the previous paragraph the ‘well defined interfaces’ of classes were discussed, ending in the conclusion that the proposition ‘well defined’ did not hold.
Section 3.3 “application driven modelling” discusses the measure of reusability in further detail. Besides that, in the object-oriented literature the problem of configuration management is rarely addressed.

Cheaper & better maintenance
The level of reusability will directly influence the cost of maintenance.
Whether the advantages hold in practice or not, given the remarks made above, there must be room for improvement.

After seeing the list of observations, the maturity of methodologies is only partially reflected by the properties of these methodologies. Perhaps the main problem is the attitude of software engineers. As long as the software community reflects their failures on the methodologies they use, and do not take full responsibility for their failures, it is unlikely that articles titled “...inheritance...considered harmful” will change anything. A discussion on the maturity of software methodologies will be marginal in effect as long as the software community is not prepared to adopt the scientific engineering principles as mentioned in the definition of software engineering.

References
5. P. Coad, E. Yourdon: Object-Oriented Analysis. (1990) Yourdon Press