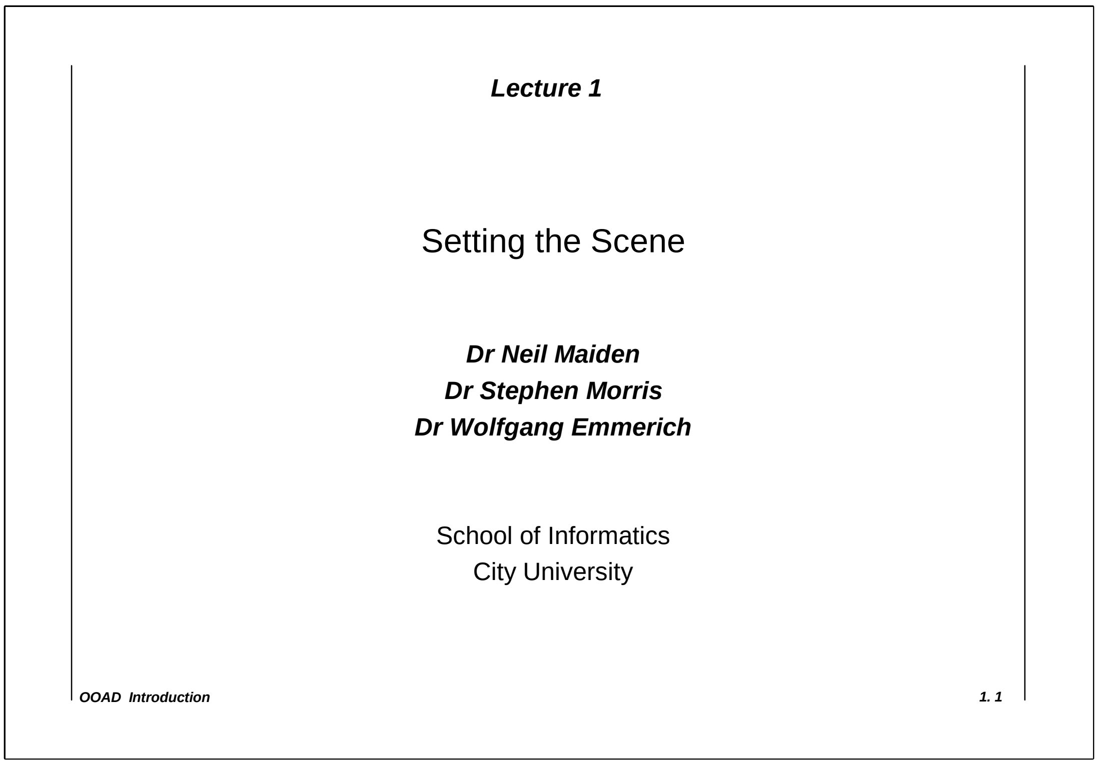


***Lecture 1***

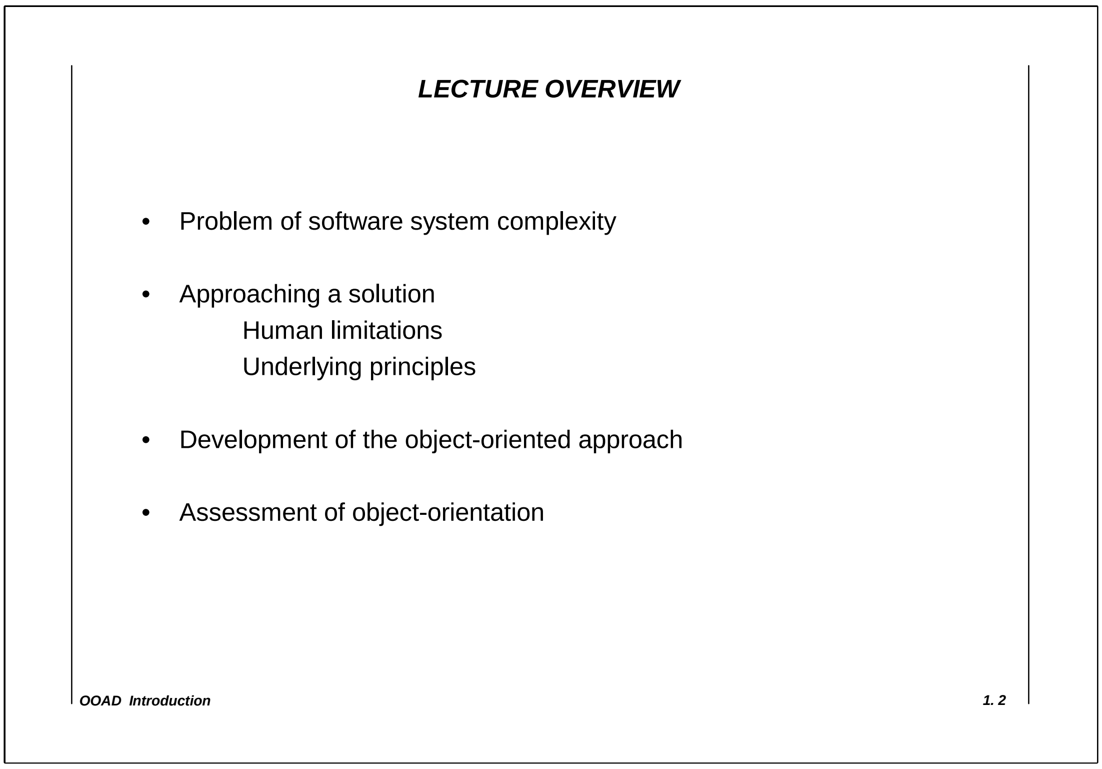
# Setting the Scene

***Dr Neil Maiden***
***Dr Stephen Morris***
***Dr Wolfgang Emmerich***

School of Informatics
City University

*OOAD Introduction* *1. 1*

The aim of this first lecture is to set the scene for this module. We shall try to theme each lecture by a question. During this session we are going to answer the question: *What is object-orientation all about, what is its evolution and how can it be positioned with respect to other (structured) methods?*



## *LECTURE OVERVIEW*

- Problem of software system complexity
- Approaching a solution
  - Human limitations
  - Underlying principles
- Development of the object-oriented approach
- Assessment of object-orientation

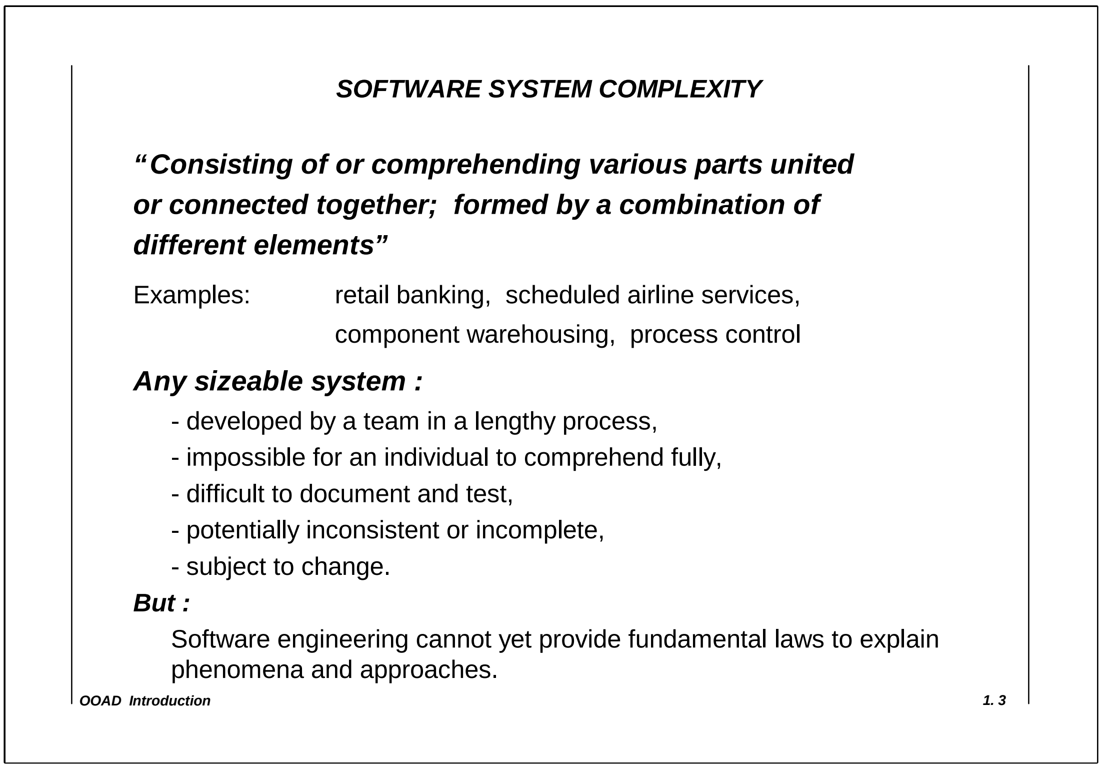
*OOAD Introduction* *1. 2*

In attempting to answer this question, we will have to look at the problem that object-orientation is trying to address. This is very much the steadily increasing complexity of software. Whe are going to look at this problem from different perspectives: We are going to discuss the reasons for software complexity, provide examples for complex software systems, and discuss common properties that any complex system has.

In attempting to find a solution to software complexity, we will have to take limitations of the human mind into account, because after all the early phases of software construction are labour intensive and still very much rely on the capabilities of the agents performing them. The solution we propose reveals a number of principles that aid the human agents performing a software process. These are the principles of *abstraction*, *hierarchy* and *decomposition*.

These principles are very nicely supported by the object-oriented approach to software development. We are then going to take a historical perspective and present the roots of objec-orientation in simulation languages, the boost the approach got from object-oriented programming languages, the ripening in object-oriented design and analysis methods. We will also briefly discuss more recent trends in object-oriented databases and open systems.

We are going to conclude this week's session with an assessment of object-orientation and identify its strengths and weaknesses.



## *SOFTWARE SYSTEM COMPLEXITY*

***"Consisting of or comprehending various parts united or connected together; formed by a combination of different elements"***

| | |
|---|---|
| Examples: | retail banking, scheduled airline services, |
| | component warehousing, process control |

***Any sizeable system :***

- developed by a team in a lengthy process,
- impossible for an individual to comprehend fully,
- difficult to document and test,
- potentially inconsistent or incomplete,
- subject to change.

***But :***

Software engineering cannot yet provide fundamental laws to explain phenomena and approaches.

*OOAD Introduction* *1. 3*

The first lines on this slide include the definition of complexity from the Oxford English Dictionary.

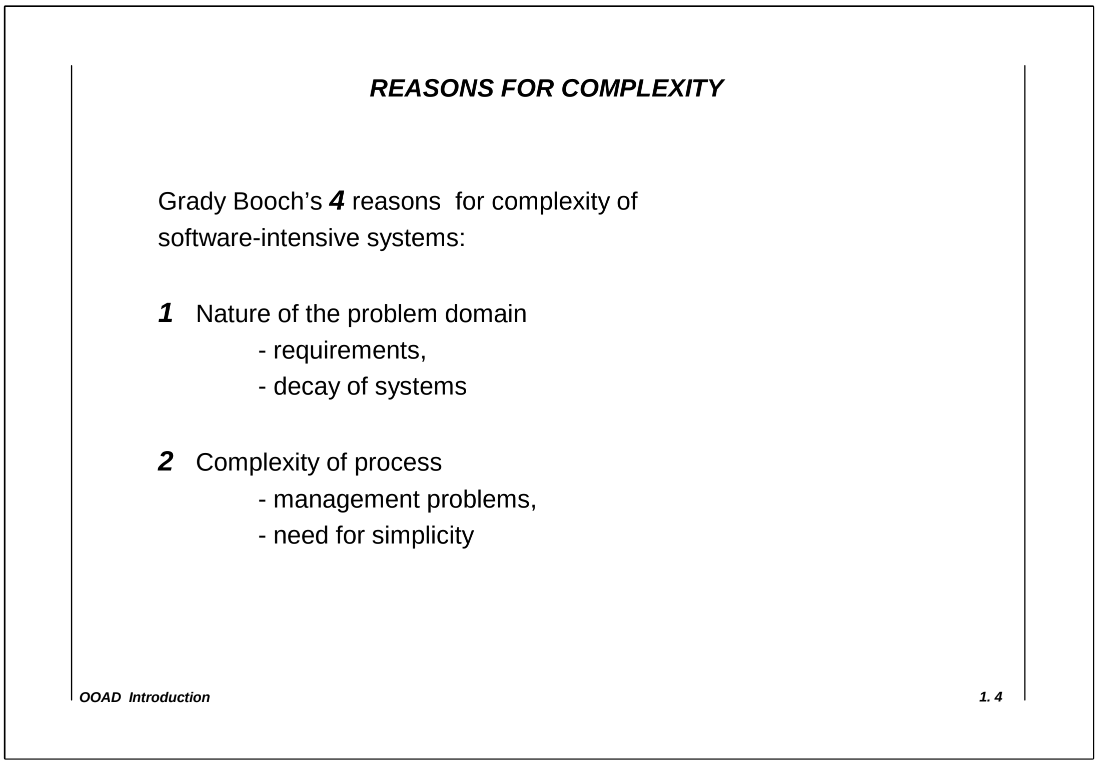
In the world of software systems, we would therefore be looking at complex examples, such as retail banking systems, mobile phone switches, airline reservation systems, component warehouse systems and process control systems of, say a nuclear power plant.

The complexity in software systems arises from a number of properties software systems have in common with any system of size. These systems are developed by a team of developers, often in a lengthy process. The size of the system is such that individuals can no longer fully comprehend the system. They are difficult to document and test. They may be inconsistent and incomplete and they radically change in order to meet changing requirements.

Please note that the complexity that we are interested in here is macro complexity, i.e. the reflection of complex processes and information of the real world in a software system, as opposed to micro complexity of algorithms that complexity theory, a research field of theoretical computer science is interested in.

The physical sciences have, in many cases, provided fundamental natural laws to explain complexity and its phenomena, eg gravitation or thermodynamics. Software engineering, by reason of its 'social' content, is in some ways closer to the social sciences and cannot yet provide any such laws.

On the next slide we are going to provide a more theoretical view of complexity from one of the founders of object-oriented methods...



## REASONS FOR COMPLEXITY

Grady Booch's ***4*** reasons for complexity of software-intensive systems:

***1*** Nature of the problem domain
- requirements,
- decay of systems

***2*** Complexity of process
- management problems,
- need for simplicity

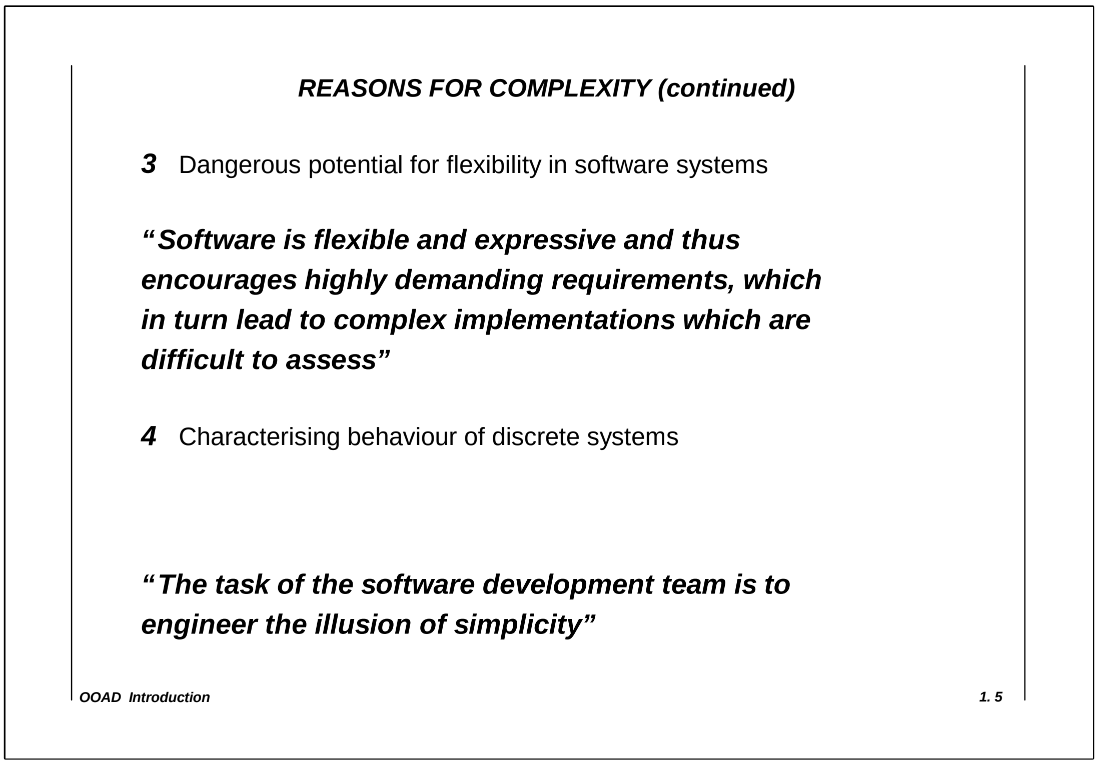
*OOAD Introduction* *1. 4*

In [Booc94], Grady Booch identifies four major reasons for complexity of any system that has an intensive software component.

The first reason is related to the application domains for which the software system is being constructed. The people who have the knowledge and skills to develop software usually do not have detailed domain knowledge and they need to acquire the requirements for the software system from that particular domain. Also these requirements are usually not stable but evolve. They evolve during the construction of the system as well as after its delivery requiring continous evolutions of the system. Complexity is often increased in trying to preserve the investments that were made in *legacy* applications. Then components addressing new requirements have to be integrated with existing legacy applications and interoperability problems caused by the heterogeneity of different system components introduces new complexity.

The second reason is the complexity of the software development process. Complex software intensive systems cannot be developed by single developers but rather require teams of developers to work on it. This adds additional overhead as the developers have to communicate about the development effort and about intermediate artefacts they produce in order to make them as consistent and complete as possible. This complexity often gets even more difficult to handle if the teams do not work in one location but are geographically dispersed. Then the management of these processes becomes an important subtask on its on and they need to be kept as simple as possible.

On the next slide, we are going to review the third and fourth reasons...


**REASONS FOR COMPLEXITY (continued)**

***3*** Dangerous potential for flexibility in software systems

***"Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to assess"***

***4*** Characterising behaviour of discrete systems

***"The task of the software development team is to engineer the illusion of simplicity"***

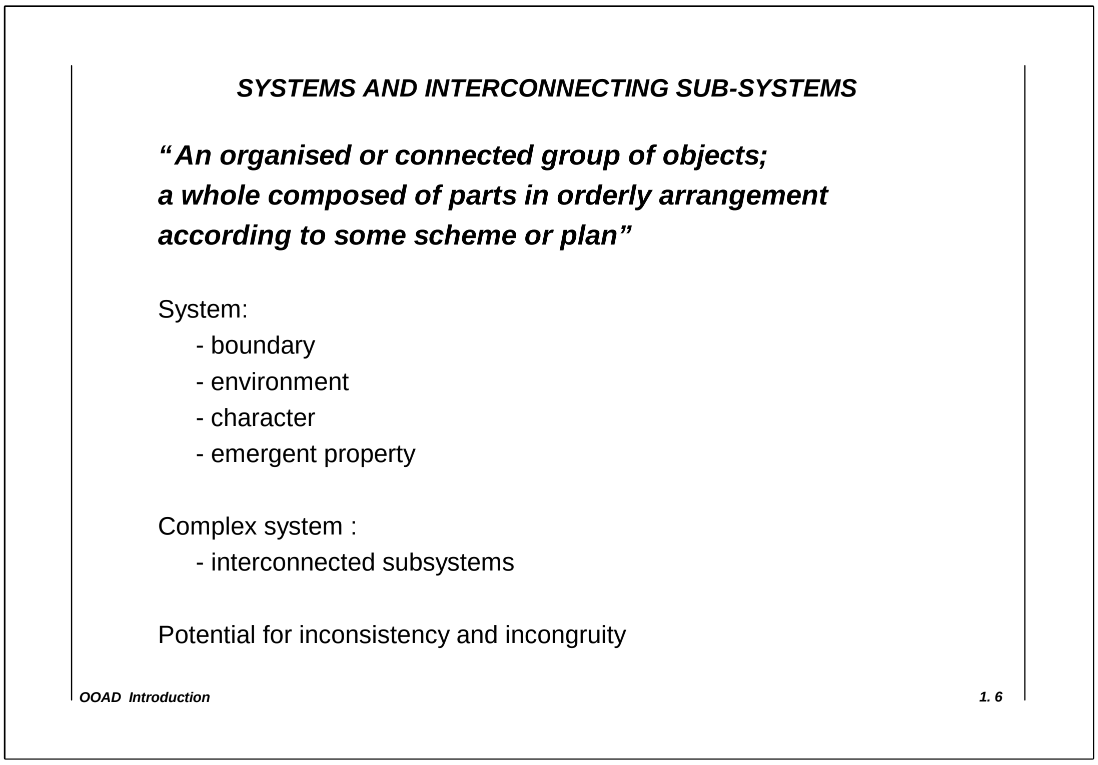
OOAD Introduction 1. 5

Booch's third reason is the danger of flexibility. Software offers a very high flexibility for changes. Hence, developers can express almost every kind of abstraction. This also often leads to situations where developers develop software components themselves rather than purchasing them from somewhere else. Unlike other industry the production depth in software is very huge. The building or automobile industry largely rely on highly specialised suppliers delivering parts and the companies just produce the design, the part specifications and assembly the parts that are delivered just in time. With software this is different and many software companies develop ever single component from scratch.

The flexibility also triggers more demanding requirements which make the products even more complicated as it is suggested by the quote on this slide, which is taken from the ESA report on the Ariane 5 failure [http:/www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html].

The final reason for complexity Booch gives is related to the difficulty in characterising the behaviour, i.e. the dynamics of a software system. While the human imagination suffices to describe static properties of systems, given they are properly decomposed, humans have problems to describe the behaviour of a complex system. This is because to describe behaviour, it is not sufficient to describe the properties of a system but the sequence of values these properties take over time needs to be specified.

The quote from [Booc94] at the bottom of the slide provide a nice conclusion of the impact complexity has for software development.



### SYSTEMS AND INTERCONNECTING SUB-SYSTEMS

***"An organised or connected group of objects;***
***a whole composed of parts in orderly arrangement***
***according to some scheme or plan"***

System:
- boundary
- environment
- character
- emergent property

Complex system :
- interconnected subsystems

Potential for inconsistency and incongruity

*OOAD Introduction* *1. 6*

Software systems are systems itself and let us now focus on the definition of a system. The quote on the top of this slide gives the definition of a system from the Oxford English Dictionary.

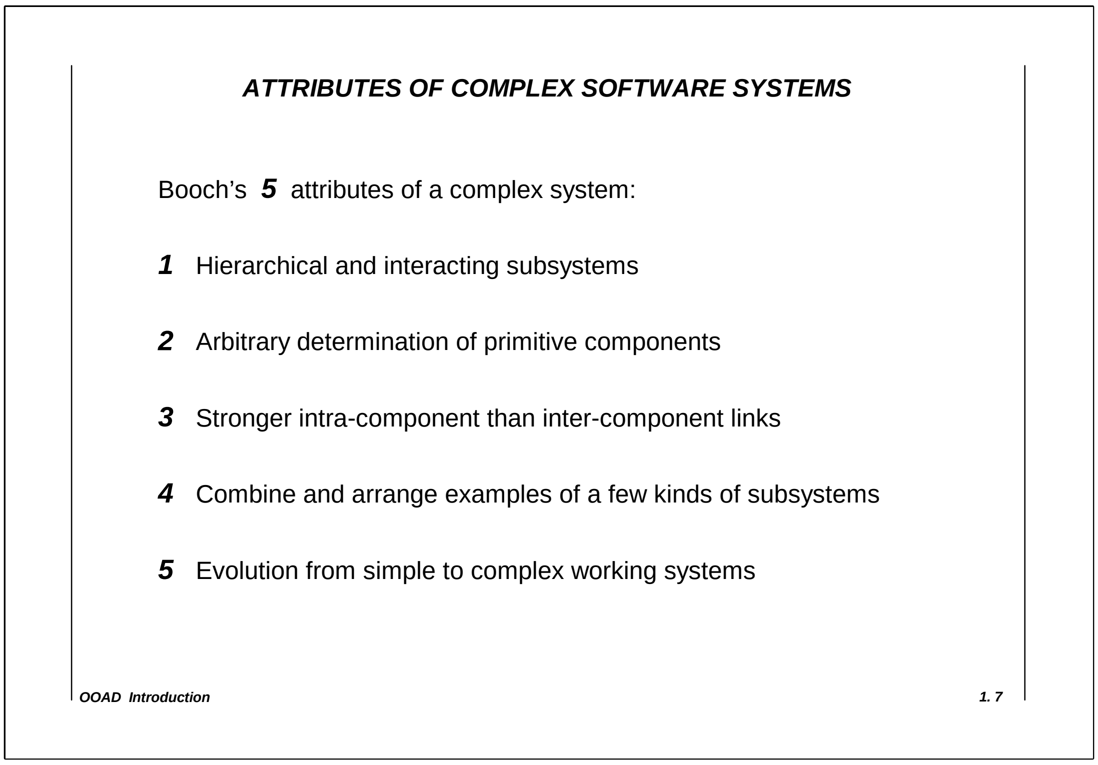
Checkland and Scholes define systems in [CS90] to have a clear boundary an embedding into its operating environment, a homogeneous character and an emergent property as a whole.

These considerations apply to software systems as well. They have to be properly bounded in the sense that it has to be defined which operations are being performed within the system and which parts are being performed without the system. They have to be properly embedded into their environment. The should have a unique character, which is often expressed in terms of the non-functional requirements the system should meet and they have an emergent property in the sense that only the whole software system renders its components useful.

Complex systems are constructed by interconnecting subsystems, which are increasingly often systems on their own rights.

Interconnecting subsystems: personal computer (I/O, processor, memory), weather system (atmosphere, oceans, land masses), local ecology (soil, buildings, micro-climate, users)

If these subsystems are constructed independently, there is a certain potential for inconsistencies and incongruities that usually are undesirable. According to Checkland and Scholes, the results of 'inconsistencies' are disasters, such as smog or pests. With software systems these inconsistencies and incongruities materialise in requirements that are not met, system malfunctions and crashes.



## ATTRIBUTES OF COMPLEX SOFTWARE SYSTEMS

Booch's **5** attributes of a complex system:

**1** Hierarchical and interacting subsystems

**2** Arbitrary determination of primitive components

**3** Stronger intra-component than inter-component links

**4** Combine and arrange examples of a few kinds of subsystems

**5** Evolution from simple to complex working systems

*OOAD Introduction* *1. 7*

Booch has identified five properties that architectures of complex software systems have in common.

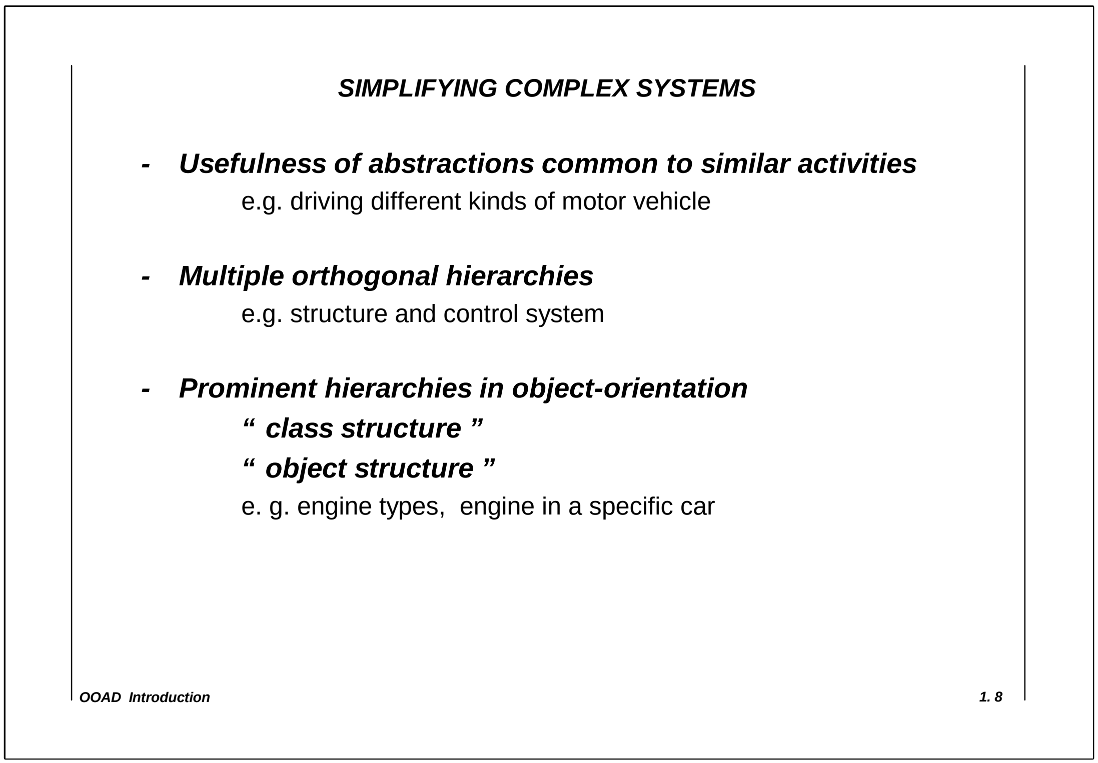
Firstly, every complex system is decomposed into a hierarchy of subsystems. This decomposition is essential in order to keep the complexity of the overall system manageable. These subsystems, however, are not isolated from each other, but interact with each other.

Very often subsystems are decomposed again into subsubsystems, which are decomposed and so on. The way how this decomposition is done and when it is stopped, i.e. which components are considered primitive, is rather arbitrary and subject to the architects decision.

The decomposition should be chosen, such that most of the coupling is between components that lie in the same subsystem and only a loose coupling exists between components of different subsystem. This is partly motivated by the fact that often different individuals are in charge with the creation and maintenance of subsystems and every additional link to other subsystems does imply an higher communication and coordination overhead.

Certain design patterns re-appear in every single subsystem. Examples are patterns for iterating over collections of elements, or patterns for the creation of object instances and the like. A collection of extremely useful patterns can be found in [GHJV94]

The development of the complete system should be done in slices so that there is an increasing number of subsystems that work together. This facilitates the provision of feedback about the overall architecture.



***SIMPLIFYING COMPLEX SYSTEMS***

- ***Usefulness of abstractions common to similar activities***
  e.g. driving different kinds of motor vehicle
- ***Multiple orthogonal hierarchies***
  e.g. structure and control system
- ***Prominent hierarchies in object-orientation***
  ***“ class structure ”***
  ***“ object structure ”***
  e. g. engine types,  engine in a specific car

*OOAD Introduction* *1. 8*

We are now going to look at whether we can regonise any discernable common forms in systems that can be used to simplify them in order to make them more manageable.

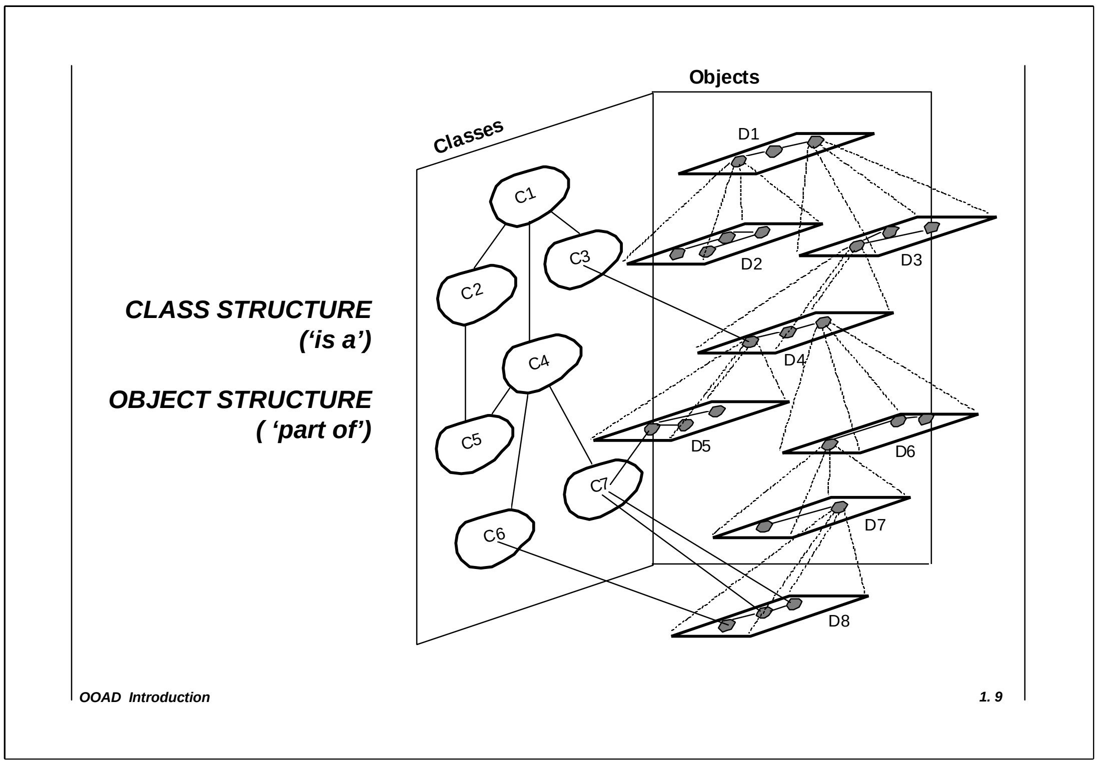
One mechanism to simplify concerns in order to make them more manageable is to identify and understand abstractions common to similar objects or activities.

We can use a car as an example (which are considerable complex systems). Understanding common abstractions in this particular example would, for instance, involve the insight that clutch, accelerator and brakes facilitate the use of a wide range of devices, namely transport vehicles depending on transmission of power from engine to wheels)

Another principle to understand complex systems is the separation of concerns leading to multiple hierarchies that are orthogonal to each other.

In the car example, this could be, for instance, the distinction between physical structure of the car (chassis, body, engine), functions the car performs (forward, back, turn) and control systems the car has (manual, mechanical, electrical).

In object-orientation, the class structure and the object structure relationship is the simplest form of related hierarchy. It forms a canonical representation for o-o analysis. The next slide attempts a visualisation of the relationship between these two hierarchies and is taken from Booch's book.



**CLASS STRUCTURE**
**(‘is a’)**

**OBJECT STRUCTURE**
**( ‘part of’)**

*OOAD Introduction* *1. 9*

As an example for these different hierarchies and their relationships, this slide represents the relationship between two different hierarchies: a hierarchy of objects and a hierarchy of classes. It is of no concern at the moment what the precise difference between a class and an object is; they will be distinguished from each other in the next week's lecture.

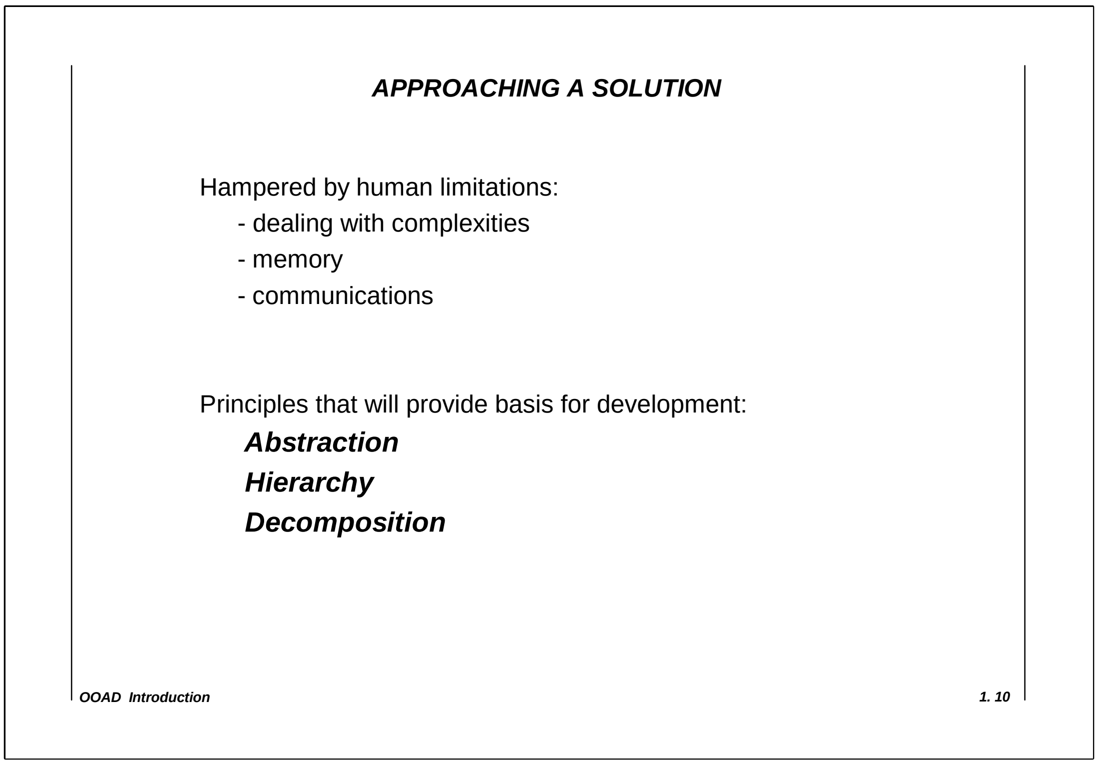
The class structure defines the 'is-a' hierarchy, identifying the commonalities between different classes at different levels of abstractions. Hence class C4 is also a class C1 and therefore has every single property that C1 has. C4, however, may have more specific properties that C1 does not have; hence the distinction between C1 and C4.

The object structure defines the 'part-of' representation. This identifies the composition of an object from component objects, like a car is composed from wheels, a steering wheel, a chassis and an engine.

The two hierarchies are not entirely orthogonal as objects are instances of certain classes. The relationship between these two hierarchies are shown by identifying the instance-of relationship as well. The objects in component D8 are instances of C6 and C7

As suggested by the diagram, there are many more objects then there are classes. The point in identifying classes is therefore to have a vehicle to describe only once all properties that all instances of the class have.

We are going to consider next, how do we approach the problem of analysing some object or situation, as yet undefined?



## APPROACHING A SOLUTION

Hampered by human limitations:
- dealing with complexities
- memory
- communications

Principles that will provide basis for development:

***Abstraction***
***Hierarchy***
***Decomposition***

OOAD Introduction 1. 10

When we devise a methodology for the analysis and design of complex systems, we need to bear in mind the limitations of human beings, who will be the main acting agents, especially during early phases.

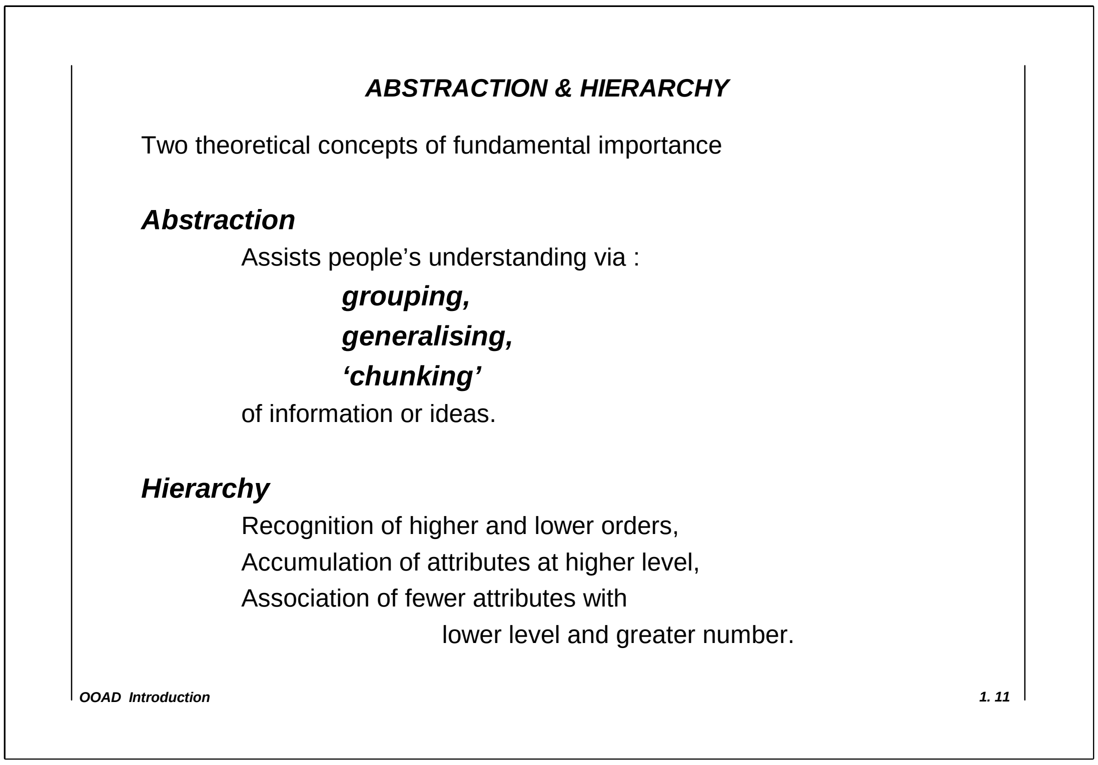
Unlike computers, human beings are rather limited in dealing with complex problems and any method need to bear that in mind and give as much support as possible. Human beings are able to understand and remember fairly complex diagrams, though linear notations expressing the same concepts are not dealt with so easily. This is why many methods rely on diagramming techniques as a basis.

The human mind is also rather limited. Miller revealed in 1956 that humans can only remember 7 plus or minus one item at once. Methods should therefore encourage its users to bear these limitations in mind and not deploy overly complex diagrams.

The analysis process is a communication intensive process where the analyst has to have intensive communications with the stakeholders who hold the domain knowledge. Also the design process is a communication intensive process, since the different agents involved in the design need to agree on decompositions of the system into different hierarchies that are consistent with each other.

Bearing in mind these limitations, these are the principles proposed for object-oriented development: abstraction, hierarchy and decomposition

We will now look at these principles in more detail...


***ABSTRACTION & HIERARCHY***

Two theoretical concepts of fundamental importance

***Abstraction***

Assists people's understanding via :

***grouping,***

***generalising,***

***'chunking'***

of information or ideas.

***Hierarchy***

Recognition of higher and lower orders,

Accumulation of attributes at higher level,

Association of fewer attributes with

lower level and greater number.

*OOAD Introduction* *1. 11*

In general abstraction assists people's understanding by grouping, generalising and chunking information.

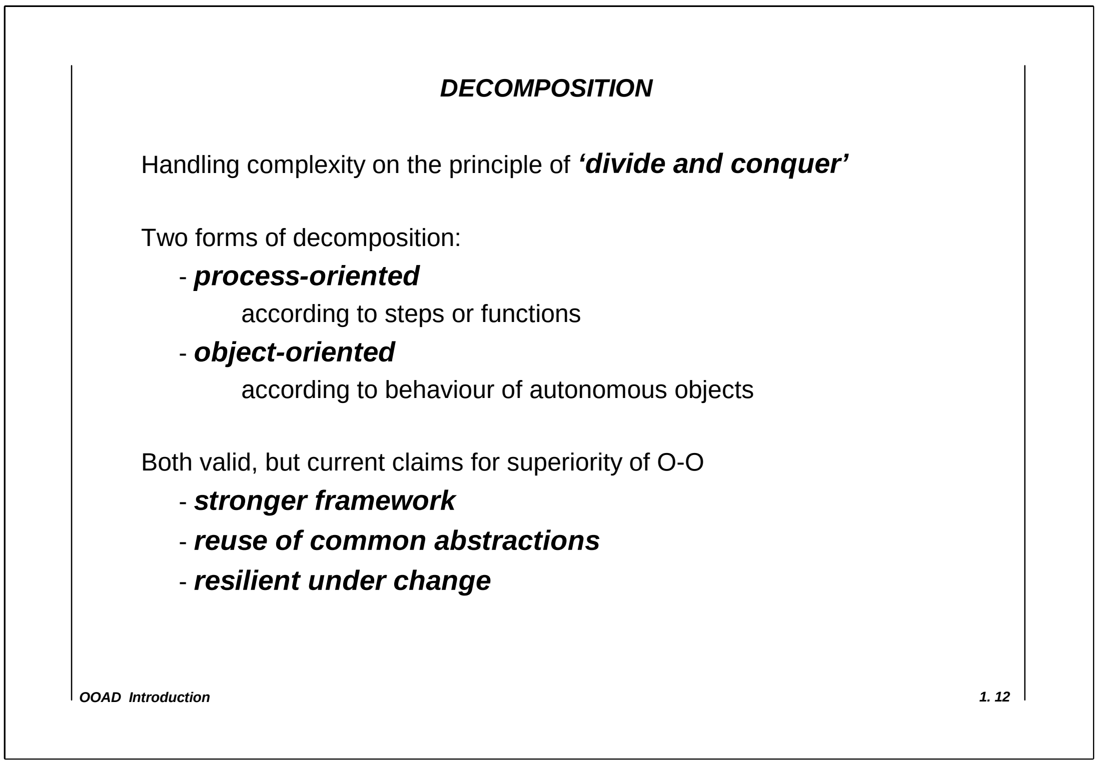
Object-orientation attempts to deploy abstraction. The common properties of similar objects are defined in an abstract way in terms of a class. Properties that different classes have in common are identified in more abstract classes and then an is-a relationship defines the inheritance between these classes.

Different hierarchies support the recognition of higher and lower orders. A class high in the is-a hierarchy is a rather abstract concept and a class that is a leaf represents a fairly concrete concept. The is-a hierarchy also identifies concepts, such as attributes or operations, that are common to a number of classes and instances thereof.

Similarly, an object that is up in the part-of hierarchy represents a rather coarse-grained and complex objects, assembled from a number of objects, while objects that are leafs are rather fine grained.

But note that there are many other forms of patterns which are non-hierarchical: interactions, 'relationships'.

Both concepts, abstraction and hierarchy are associated, in practice, with decomposition as it is shown on the next slide...



***DECOMPOSITION***

Handling complexity on the principle of ***'divide and conquer'***

Two forms of decomposition:

- ***process-oriented***
  according to steps or functions
- ***object-oriented***
  according to behaviour of autonomous objects

Both valid, but current claims for superiority of O-O

- ***stronger framework***
- ***reuse of common abstractions***
- ***resilient under change***

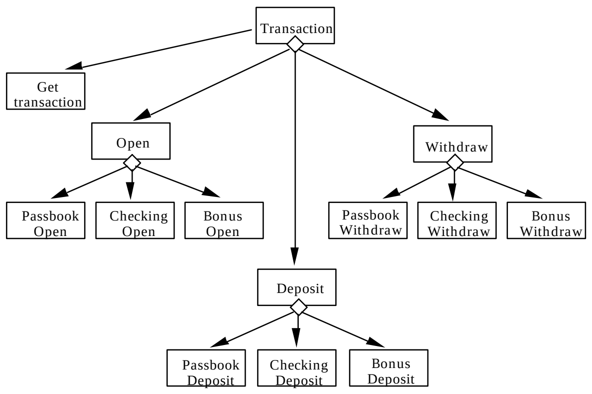
*OOAD Introduction* *1. 12*

Decomposition is an important technique for coping with complexity based on the idea of divide and conquer. In dividing a problem into a subproblem the problem becomes less complex and easier to overlook and to deal with. Repeatedly dividing a problem will eventually lead to subproblems that are small enough so that they can be conquered. After all the subproblems have been conquered and solutions to them have been found, the solutions need to be composed in order to obtain the solution of the whole problem.

The history of computing has seen two forms of decomposition, process-oriented and object-oriented decomposition. Process-oriented decompositions divide a complex process, function or task into simpler subprocesses until they are simple enough to be dealt with. The solutions of these subfunctions then need to be executed in certain sequential or parallel orders in order to obtain a solution to the complex process. Object-oriented decomposition aims at identifying individual autonomous objects that encapsulate both a state and a certain behaviour. Then communication among these objects leads to the desired solutions.

Although both solutions help dealing with complexity we have reasons to believe that an object-oriented decomposition is favourable because, the object-oriented approach provides for a semantically richer framework that leads to decompositions that are more closely related to entities from the real world. Moreover, the identification of abstractions supports (more abstract) solutions to be reused and the object-oriented approach supports the evolution of systems better as those concepts that are more likely to change can be hidden within the objects.

## *A FUNCTION / DATA DECOMPOSITION*

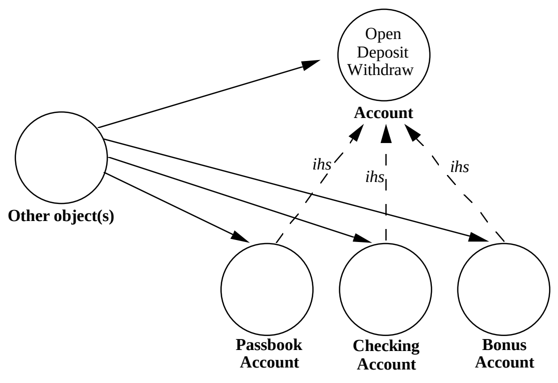


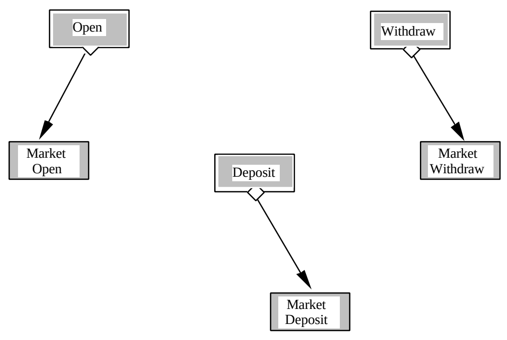
***OOAD Introduction*** ***1. 13***

To illustrate the three concepts of abstraction, hierarchy and decomposition and validate the claim that object-orientation is favourable, consider an example used by Jacobson (pp 135-141).

As this example is discussed in depth in the course text, we refrain from providing detailed notes for the next four slides, including this one.

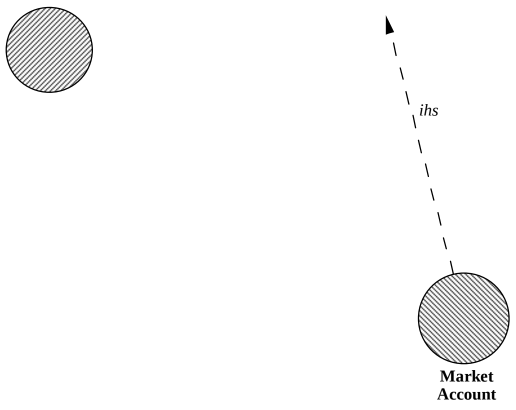
## A OBJECT-ORIENTED DECOMPOSITION



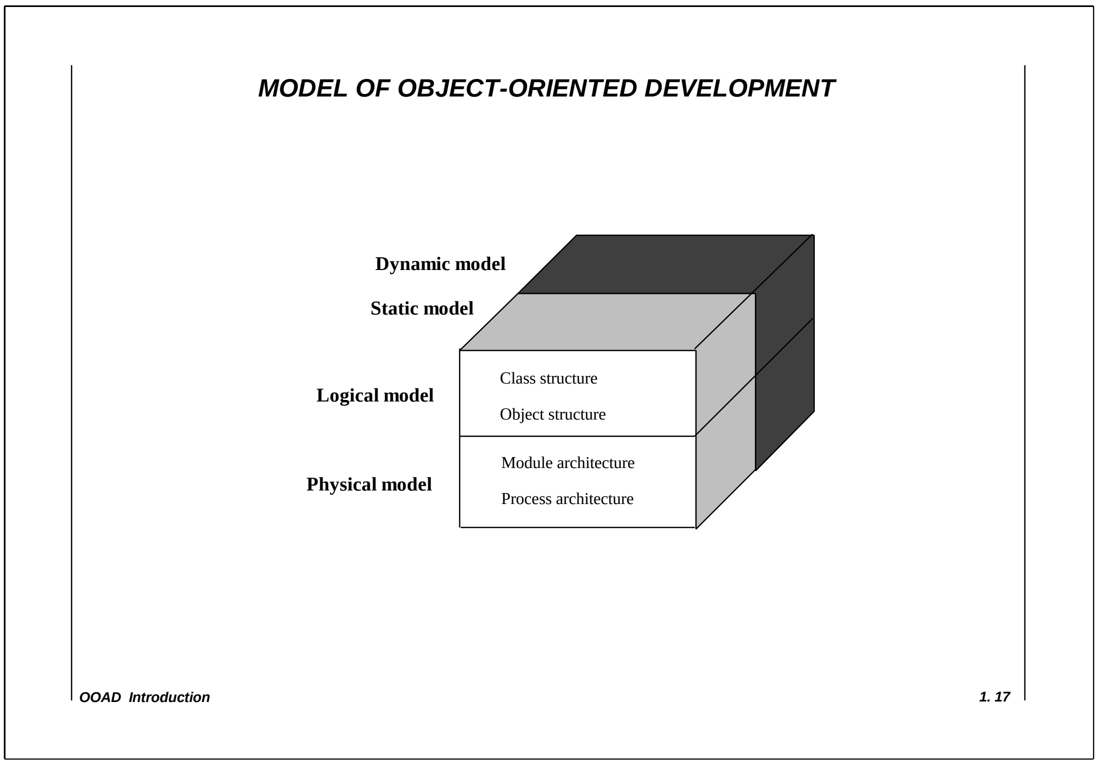


***Overlay for***
***FUNCTION / DATA DECOMPOSITION***

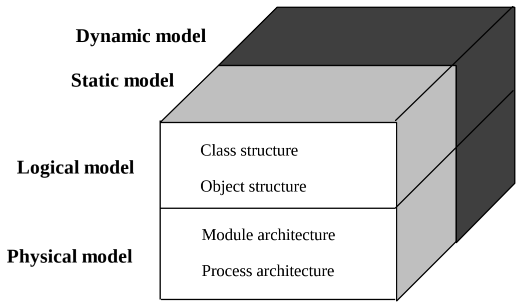
*OOAD Introduction* *1. 15*



***Overlay for***
***OBJECT-ORIENTED DECOMPOSITION***



## MODEL OF OBJECT-ORIENTED DEVELOPMENT



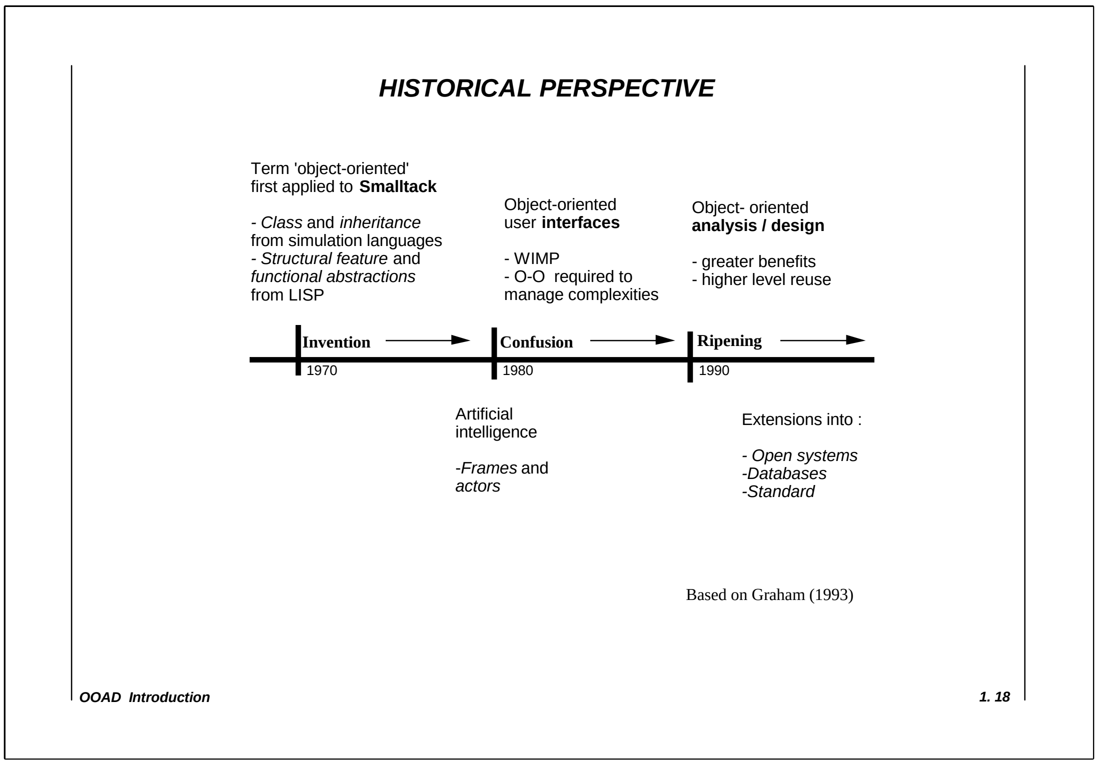
OOAD Introduction 1. 17

Booch presents a model of object-oriented development that identifies several relevant perspectives.

The classes and objects that form the system are identified in a logical model. For this logical model, again two different perspectives have to be considered. A static perspective identifies the structure of classes and objects, their properties and the relationships classes and objects participate in. A dynamic model identifies the dynamic behaviour of classes and objects, the different valid states they can be in and the transitions between these states.

Besides the logical model, also a physical model needs to be identified. This is usually done later in the system's lifecycle. The module architecture identifies how classes are kept in seperately compileable modules and the process architecture identifies how objects are distributed at run-time over different operating system processes and identifies the relationships between those. Again for this physical model a static perspective is defined that considers the structure of module and process architecture and a dynamic perspective identifies process and object activation strategies and inter-process communication.

Object-orientation has not, however, emerged fully formed. In fact it has developed over a long period, and continues to change. We will briefly sketch the history of object-orientation on the next slide.



## *HISTORICAL PERSPECTIVE*

Term 'object-oriented'
first applied to **Smalltack**

- *Class* and *inheritance* from simulation languages
- *Structural feature* and *functional abstractions* from LISP

Object-oriented
user **interfaces**

- WIMP
- O-O required to manage complexities

Object- oriented
**analysis / design**

- greater benefits
- higher level reuse

**Invention** → 1970 **Confusion** → 1980 **Ripening** → 1990

Artificial
intelligence

-*Frames* and *actors*

Extensions into :

- *Open systems*
-*Databases*
-*Standard*

Based on Graham (1993)

***OOAD Introduction*** ***1. 18***

Object-orientation emerged from concepts developed in the late 60s in simulation and functional programming languages. Simula-67 was the first programming language that include the concepts of classes and inheritance relationship.

In the 70s the ideas of information hiding [Parn72] and abstract data types [LZ74] evolved from work done by David Parnas and Barbara Liskov. Liskov´s seminal paper on abstract data types created algebraic specifications, still an area of active research in theoretical computer science.

The term 'object-orientation' was first used together with Simula-80, developed by Adele Goldberg [Gold85] at Xerox Parc for the efficient and effective implementation of graphical user interfaces. Smalltalk is a pure object-oriented programming language where "everything is an object". It influenced the introduction of object-oriented concepts into imperative languages, such as Modula-2 (leading to Oberon) and C (leading to C++) and Pascal (leading to Object Pascal and Delphi).

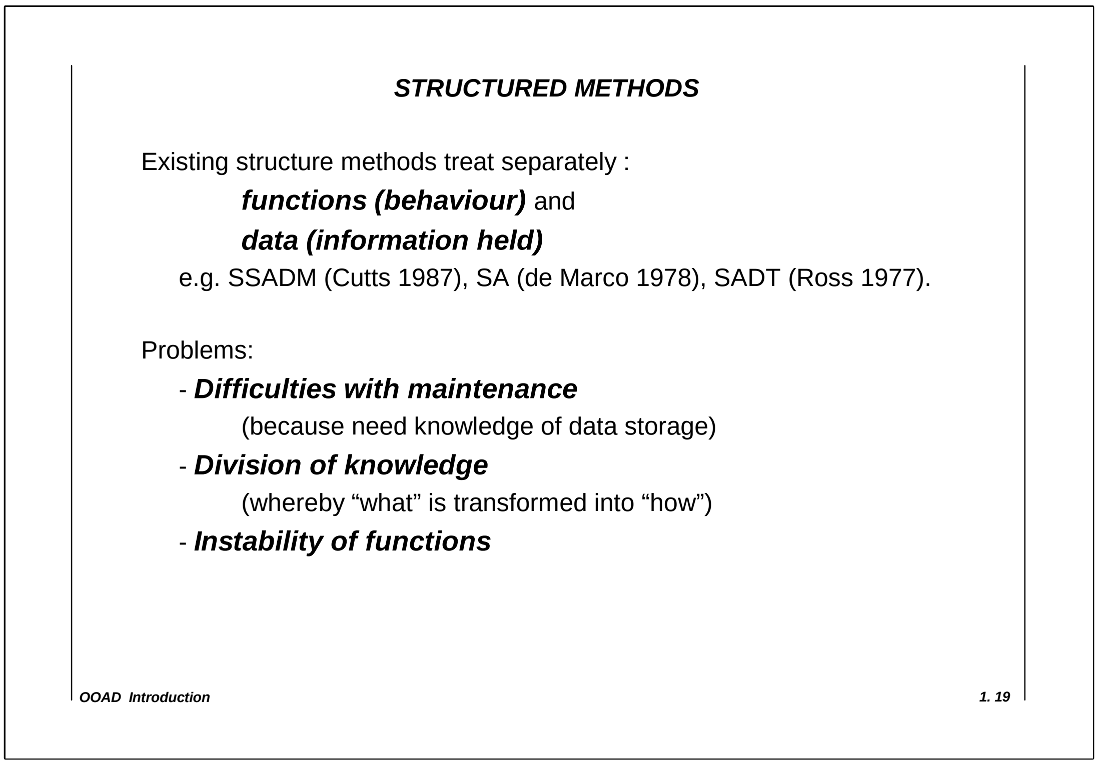
In the late 80s work started to bring object-oriented principles from programming to earlier phases, namely requirements analysis and design. These so called object-oriented methods benefit from the lack of an impedance mismatch between structural design techniques and programming languages.

Concurrently, the success of OO programming languages influenced the design of new types of databases that do not store data in tables but rather store it in objects. These so-called object databases have now become widely available.

Recent trends are set by the Object Management Group (OMG), a consortium of more than 700 leading vendors that standardises object technology, most notably the common object request broker architecture.

From this perspective object-orientation can be seen as a set of concepts that continues to develop. In the latest stage the emphasis has shifted away from clarification of basic ideas and towards standardisation of their representation and use. Hence the central element of of this course, OOSE with UML.

Object-orientation is also not the only formalised and structured approach to system development as can be seen on the next slide...



## *STRUCTURED METHODS*

Existing structure methods treat separately :

***functions (behaviour)*** and

***data (information held)***

e.g. SSADM (Cutts 1987), SA (de Marco 1978), SADT (Ross 1977).

Problems:

- ***Difficulties with maintenance***
  (because need knowledge of data storage)
- ***Division of knowledge***
  (whereby “what” is transformed into “how”)
- ***Instability of functions***

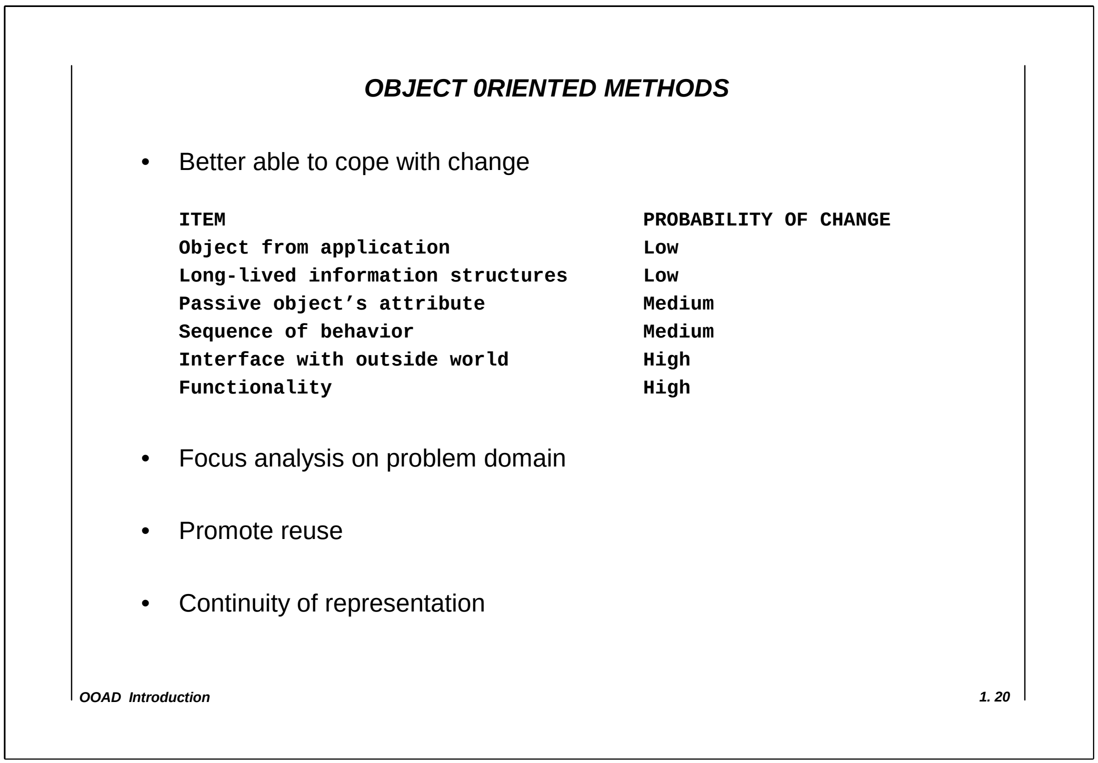
*OOAD Introduction* *1. 19*

This and the next two slides compare object-oriented and structured methods. This comparison has already been touched on from a more general perspective when we compared functional and object-oriented decompositon but now we shall try to get it down to the point.

The problem with structure-oriented methods, such as SSADM, Structured Analysis or SADT, is that they treat functions (i.e. the behaviour) of the system differently from the data (i.e. the information held somewhere within the system).

This complicates maintenance and the evolution of a system as both data and functions need to be changed. Moreover it is more difficult to isolate changes. If a certain aspect has to be changed, this almost certainly involves both the change of data structures and of algorithms. Finally the change of algorithms and data structures in structural methods often involves a number of subsequent changes to places where these data structures are used as well.

Object-oriented decomposition, on the other hand has evolved from the idea of information hiding which significantly contributes to the changeability of the system as motivated on the next slide...



***OBJECT 0RIENTED METHODS***

- Better able to cope with change

| ITEM | PROBABILITY OF CHANGE |
|---|---|
| Object from application | Low |
| Long-lived information structures | Low |
| Passive object’s attribute | Medium |
| Sequence of behavior | Medium |
| Interface with outside world | High |
| Functionality | High |

- Focus analysis on problem domain
- Promote reuse
- Continuity of representation

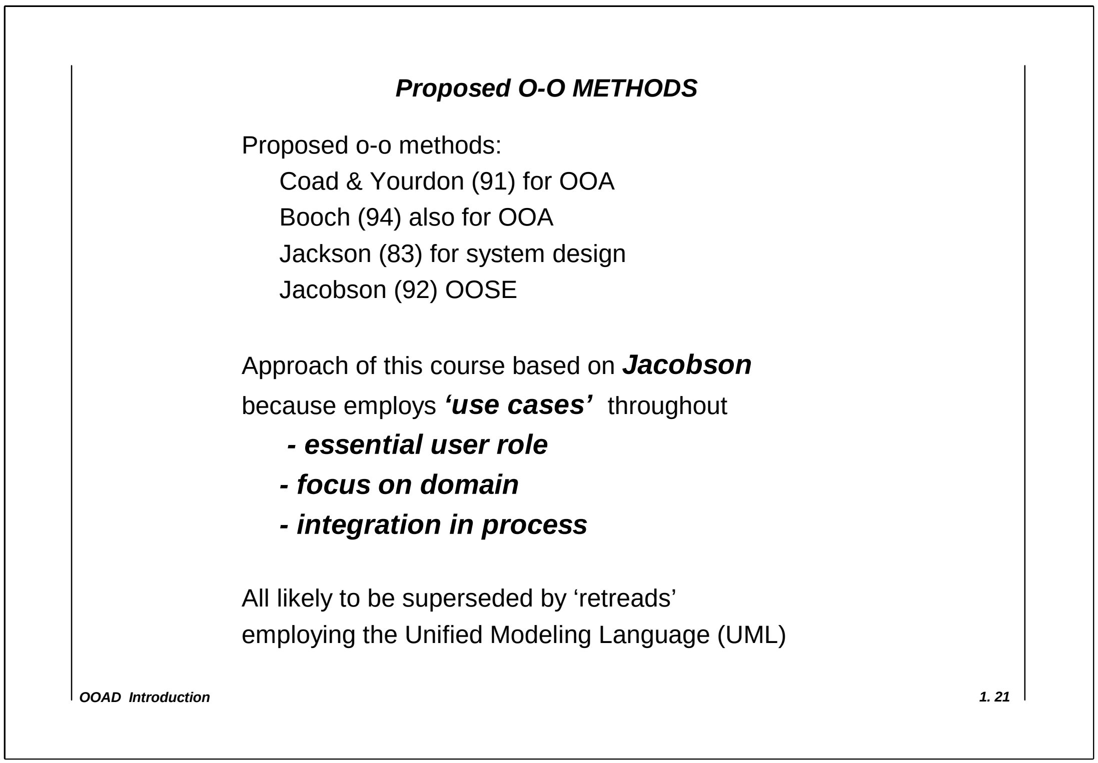
*OOAD Introduction* *1. 20*

Object-oriented decompositions of systems tend to be better able to cope with change. This is because they manage to encapsulate those items that are likely to change (such as functionality, sequence of behaviour and attributes) within an object and hide them from the outside world. This provides the advantage that the outside cannot see them and therefore cannot be dependent on them and does not need to be changed if these items change.

Also object-oriented decompositons are closer to the problem domain, as they directly represent the real-world entities in their structure and behaviour.

The abstraction primitives built into reuse have a huge potential of reuse as commonalities between similar objects can be factored out and then the solutions can be reused.

Finally, object-orientation has the advantage of continuity througout analysis, design implementation and persistent representation.



## *Proposed O-O METHODS*

Proposed o-o methods:
- Coad & Yourdon (91) for OOA
- Booch (94) also for OOA
- Jackson (83) for system design
- Jacobson (92) OOSE

Approach of this course based on ***Jacobson***
because employs ***'use cases'*** throughout
- ***- essential user role***
- ***- focus on domain***
- ***- integration in process***

All likely to be superseded by 'retreads'
employing the Unified Modeling Language (UML)

*OOAD Introduction* *1. 21*

Unfortunately, there is no single object-oriented analysis and design method that we could readily teach you and this slide lists some of the proposed methods.

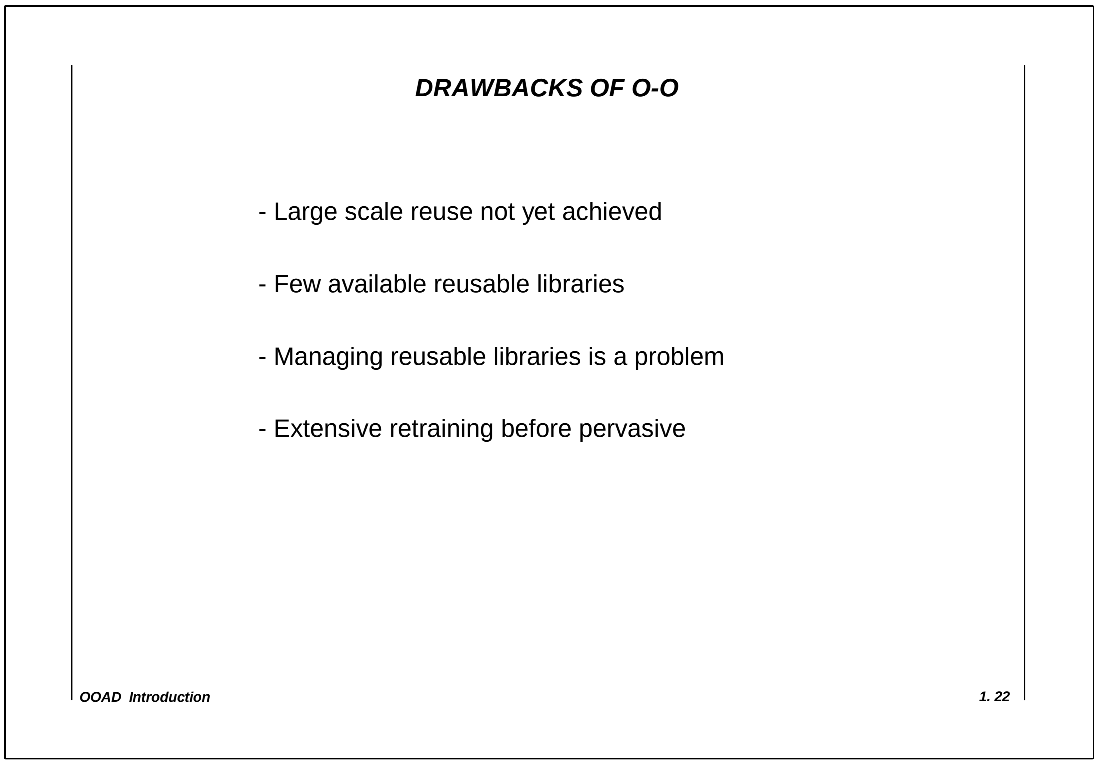
For this course we have selected Ivar Jacobson´s object-oriented software engineering approach because it supports use case scenarios.

We largely agree with industry that these scenarios are particularly useful during the elicitation of complete user reqirements. OOSE has domain focus built into it and is integrated in the process.

We should, however, note that we are going to use the unified modelling language notation. UML is currently being developed at Rational, a major consulting company in the US and also the vendor of the market leading OOAD environment. Grady Booch, Ivar Jacobson and James Rumbaugh are jointly working on the modelling language. While we are giving this lecture, UML is being evaluated by the Object Management Group to become the de-facto industry wide notation for object-oriented modelling.

It is very likely that the methods presented so far in the various books identified at the beginning will be revisited by their authors and be expressed in terms of the Unified Modelling Language.

Although UML is an important consolidation step forward in the maturation process of object-orientation, it will not be a silver bullet. As the next slide suggests, there are also problems inherent to UML.



## DRAWBACKS OF O-O

- Large scale reuse not yet achieved
- Few available reusable libraries
- Managing reusable libraries is a problem
- Extensive retraining before pervasive

OOAD Introduction 1. 22

Although object-orientation is very favourable to reusing requirements, parts of the design and implementation, large-scale reuse has not yet been achieved. We believe that this is not necessarily only a problem of object-orientation, but also of the mindset of many software professionals that do not believe in anything they have not developed themselves.

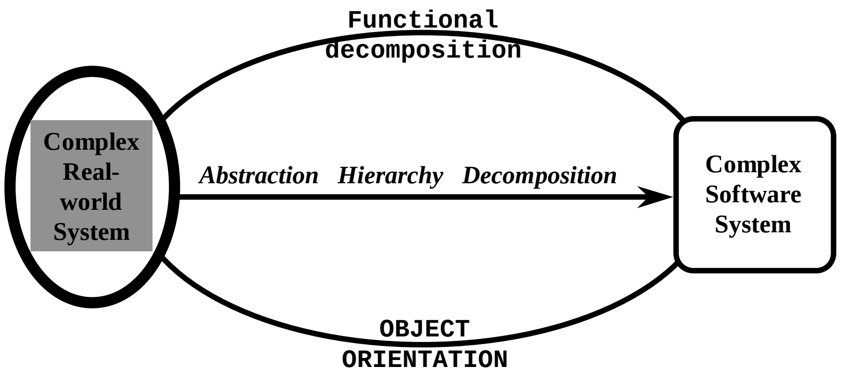
As there is little demand for reuseable components a market for components has not yet been established. Vendors are scattered and their products are rarely standardised and therefore are not exchangeable. A notable exception is the standard template library that has been standardised last year by the ANSI.

With the possibility of deploying previously developed components from reuse library, whether bought off the shelve or built inhouse, the problem of configuration management arises which has not yet been fully understood.

Also a considerable amount of retraining of staff is required before object-oriented projects start to fly and industry is still in the process of building up an experience base.

As a member of a development team (whether student, academic or commercial) you may have your mind made up for you, but essential to recognise true status of ideas and techniques.

## *Summary*



*OOAD Introduction* *1. 23*

This picture summarises the principles that we have outlined in this week's session. As software engineers, we are interested in finding principles for the mapping of complex real-world systems into supportive complex software systems.The principles we have suggested this week are abstraction, hierarchy and decomposition. They are deployed in both functional and object-oriented methods, but the latter seem to be favourable due to their support for change.

The next lecture will return to principles of object-orientation und discuss them at a sufficient level of detail.

Your first tutorial addresses the principles of abstraction, hierarchy and decomposition.

The following literature has been referenced throughout the notes for this week, which we would recommend as additional background reading.

[CS90] P. Checkland and J. Scholes: Soft systems methodology in action. Wiley, 1990.

[Gold85] A.Goldberg: The Language and its Implementation. Addison-Wesley, 1985.

[GHJV94] E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[LZ74] B.Liskov and S.Zilles: Programming with Abstract Data Types, SIGPLAN Notices 9(4):50-55, 1974.

[Parn72] D.C.Parnas: A Technique for the Software Module Specification with Examples. CACM 15(5):330-336, 1972.