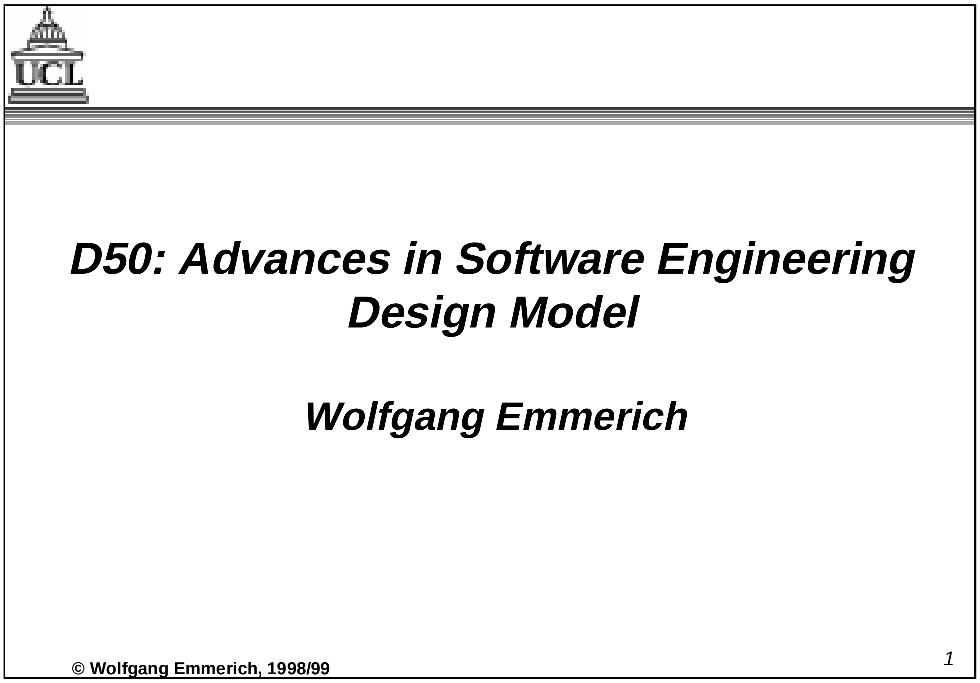
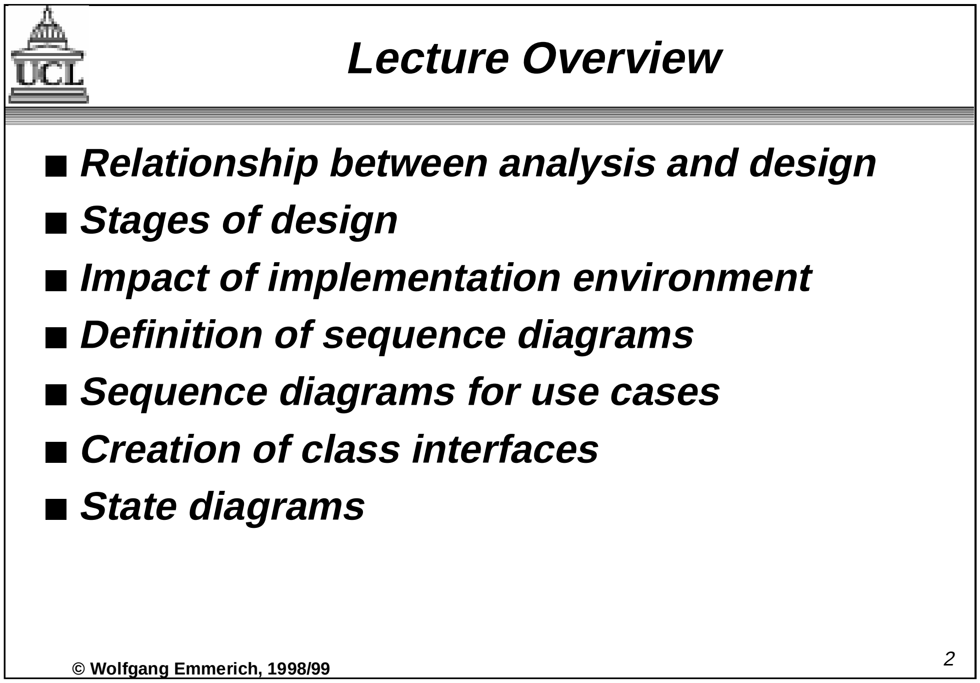


D50: Advances in Software Engineering Design Model

Wolfgang Emmerich

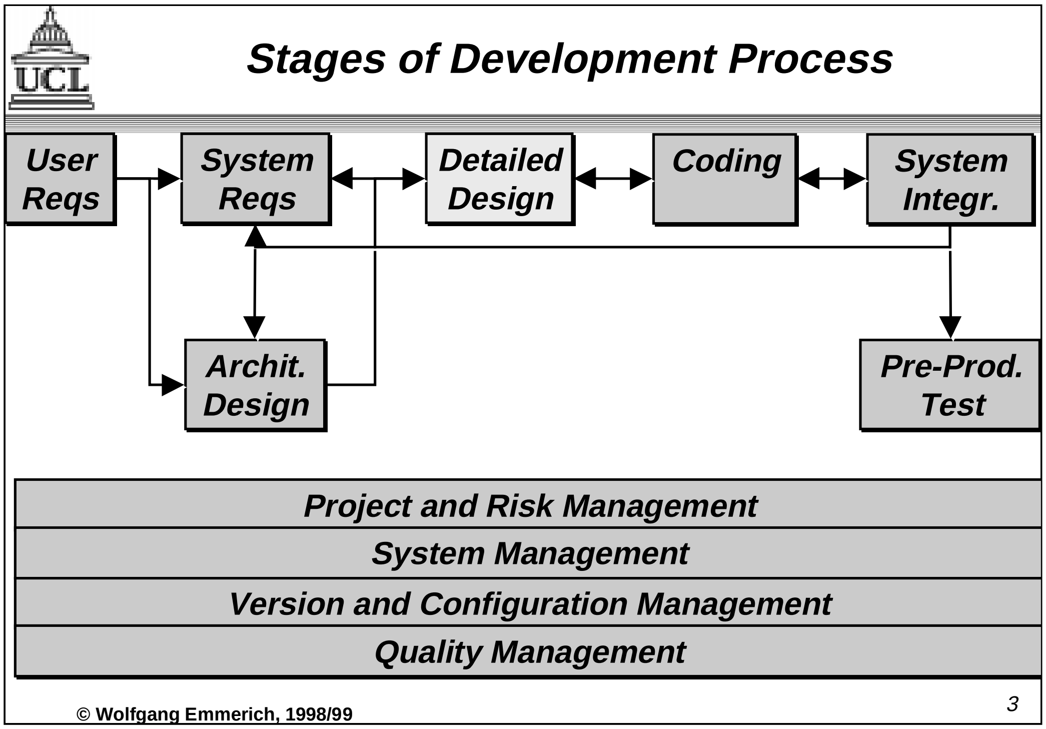
- Last week, we have discussed the derivation of an hierarchical class diagram from use cases and a list of domain objects that were produced during requirements modelling. The class diagram produced during analysis gives a logical perspective on objects in the problem domain. The classes were directly derived from problem domain objects and use cases. Moreover, we identified different roles for the classes. Interface classes were identified that take input from actors, be they humans or other programs. Entity classes model how state information is represented for the various domain objects and control classes were included as glue between interface and entity classes.
- The class diagram produced during the analysis stage, however, neglects important properties, for instance of the programming language that is used for implementing the classes. Constraints introduced on classes by the programming language and other influencing factors, therefore, need to be taken into consideration in order to compose an executable system.
- Moreover, the analysis class diagram is still rather abstract on operations. It identifies associations between clases, but does not indicate which operations use these associations in order to stimulate execution of other classes' operations. It also does not give indications on the use of other operations in the algorithms that define how operations are implemented. Without formulating these algorithms, we cannot be certain that the class diagram is complete, i.e. includes all necessary operations.
- Hence the question that we are going to answer in this lecture is: *How is the result of the analysis class diagram transformed into an implementable class diagram and how do we specify dependencies between operations*.



Lecture Overview

- ■ *Relationship between analysis and design*
- ■ *Stages of design*
- ■ *Impact of implementation environment*
- ■ *Definition of sequence diagrams*
- ■ *Sequence diagrams for use cases*
- ■ *Creation of class interfaces*
- ■ *State diagrams*

- In this lecture, we are again following the model-based approach of Jacobson, though we aim at giving a more specific step-by-step approach, therefore we will be showing again the ‘road map’ relating the two views.
- The primary purpose of design stage is to translate and complete the ‘logical’ model provided by the analysis stage into a more concrete level of abstraction, which reflects the implementation environment and can be implemented directly in code.
- During the design stage we use two important new representations, first the ‘sequence diagram’ showing the interaction of a set of objects in temporal order. In OOSE this set of objects belongs, most importantly, to a particular use case.
- The principal design element is the class, synonymous with a block in OOSE, and the operations incorporated into it complete the picture of its export interface (its outside view). In addition to the operations that were identified in the analysis stage, operations to be included into the design class diagram are derived, principally, from sequence diagrams.
- The second new representation, the ‘state diagram’, describes the temporal evoluation of an object of a given class in response to other objects inside or outside the system. This representation will be the principal subject of the second lecture on design that we will give next week.



Stages of Development Process

User Reqs

System Reqs

Detailed Design

Coding

System Integr.

Archit. Design

Pre-Prod. Test

Project and Risk Management

System Management

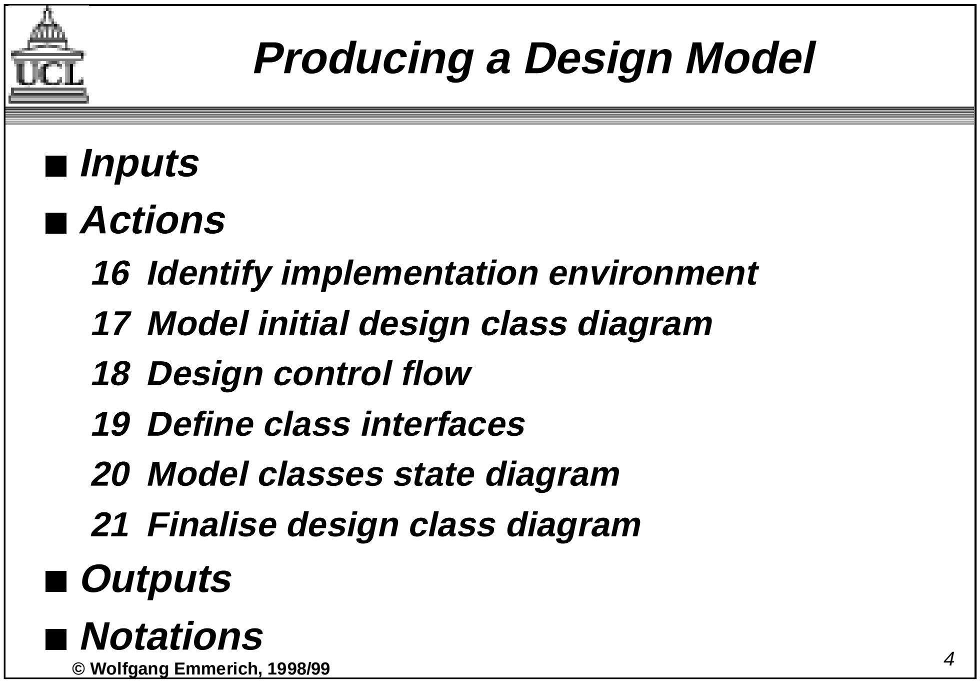
Version and Configuration Management

Quality Management

© Wolfgang Emmerich, 1998/99

3

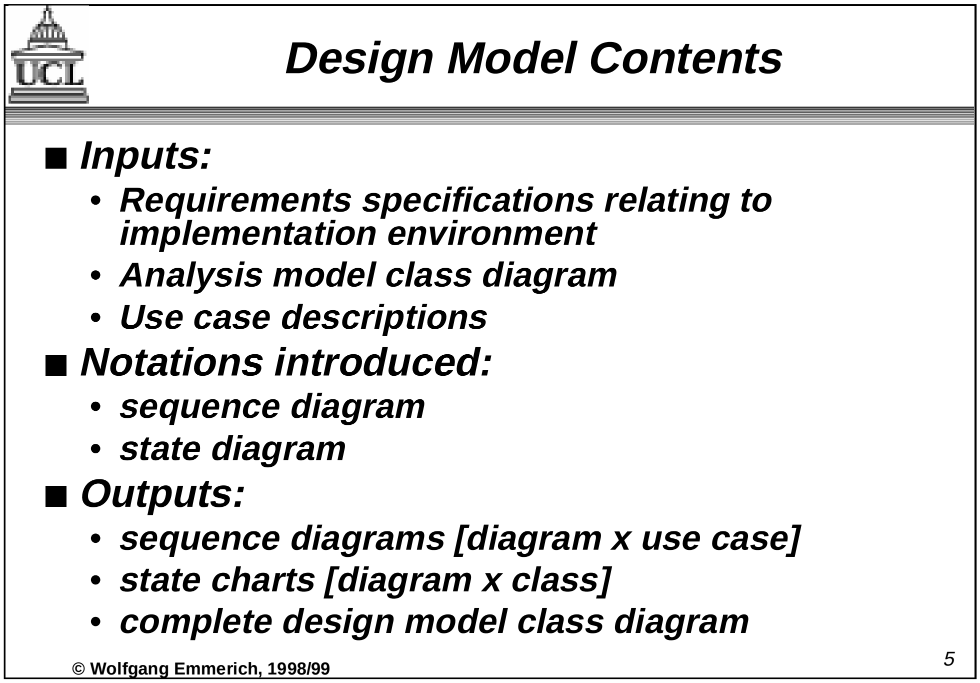
- This slide shows our ‘road map’ again in order to highlight the aspects we discuss in this lecture and indicate how they fit into the overall object-oriented development process.
- Last week, we have shown how a class diagram is derived that is hierarchically organised into nested packages. We have also indicated how the use case description that was produced in the requirements stage is elaborated and formulated in terms of domain objects and their operations. The design stage starts from these two representations.
- The broken line between analysis and design marks the essential boundary between the conclusion of the analysis phase and the beginning of a detailed design that takes into account the implementation environment, as defined in the initial requirements documents (not shown).
- This week we are going to introduce sequence diagrams that show for each use case the order in which objects send messages to other objects in order to stimulate operation executions. This exercise will reveal missing operations in the class diagram that will be added to the design class diagram.
- Let us now look at the first steps that are necessary during the design



Producing a Design Model

- **Inputs**
- **Actions**
  - *16 Identify implementation environment*
  - *17 Model initial design class diagram*
  - *18 Design control flow*
  - *19 Define class interfaces*
  - *20 Model classes state diagram*
  - *21 Finalise design class diagram*
- **Outputs**
- **Notations**

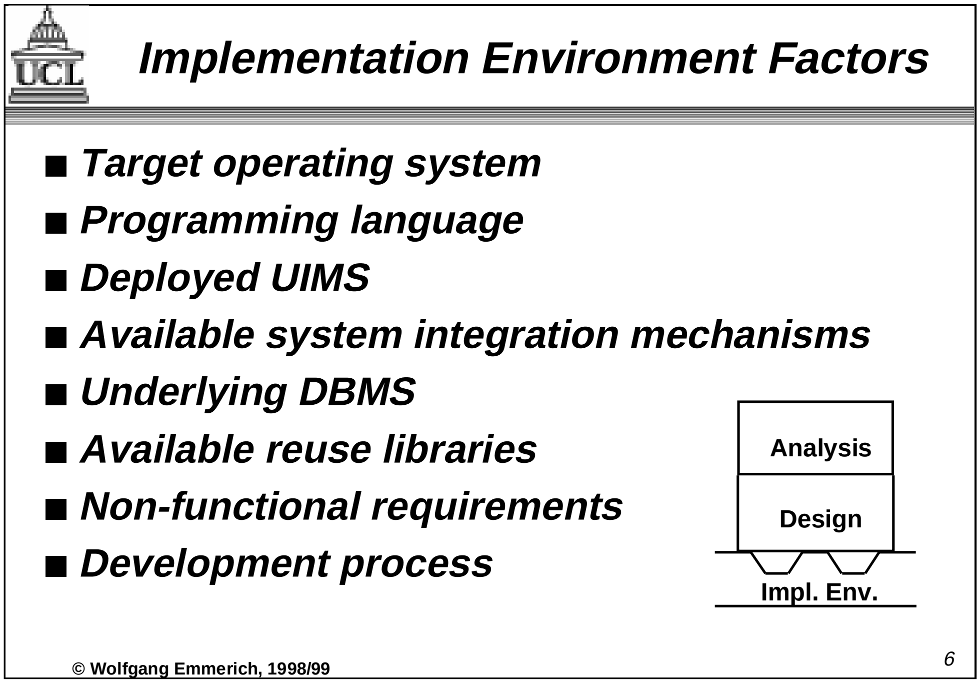
- The first step in the design stage is the identification of the implementation environment. The influencing components of the environment are, for instance, the target programming language, the user interface management system, class libraries that are available for reuse, distribution infrastructures and databases for persistent storage of entity objects.
- The second design step is the translation of the analysis class diagram into a design model class diagram. This requires revisions to make the class diagram implementable and cohesive at architectural level. Unfortunately, we cannot discuss this step in sufficient detail here because a full appreciation requires in-depth understanding of object-oriented programming languages, distributed object infrastructures, user interface construction and object databases.
- Designing the flow of control involves the determtion of messages that are passed between objects. This is done using 'sequence diagrams' for each use case.
- With the help of sequence diagrams we can then detect missing operations and complete the operations defined for classes. Also the sequence diagrams enable us to check which parameters are passed along with messages and we can check the available operations against that. Hence, sequence diagrams enable us to complete the outside view, the 'export interface' of each class.
- In next lecture we will consider in detail 'state machines' and 'state diagrams', powerful concepts in their own right, used for the behavioural design of the class.
- The final product is the design class diagram (in 'packaged' form) which will be the basis for direct implementation in code.



Design Model Contents

- ***Inputs:***
  - ***Requirements specifications relating to implementation environment***
  - ***Analysis model class diagram***
  - ***Use case descriptions***
- ***Notations introduced:***
  - ***sequence diagram***
  - ***state diagram***
- ***Outputs:***
  - ***sequence diagrams [diagram x use case]***
  - ***state charts [diagram x class]***
  - ***complete design model class diagram***

- The design takes its inputs not only from the immediately preceding analysis stage, but also from the initial set of requirements specifications. These initial documents will include basic information about the implementation environment.
- The design stage heavily uses class diagrams that were introduced already. The notation used for the class diagram in the analysis phase does not differ from the one used for the dsign phase. The contents of the design class diagram, however, might be considerably different from the one produced in the analysis stage.
- The design stage uses two new types of diagrams, sequence diagrams and state diagrams. The former is introduced in this lecture and the latter will be introduced next week. The UML provides notations for both the new types of diagram. The sequence diagram in UML is very close to the ‘interaction digram’ in Jacobson's OOSE. The origin of the state diagram is in Rumbaugh's OMT method and Rumbaugh himself borrowed many concepts from David Harel's state charts [Hare87].
- The most important output of the design stage will be a class diagram that is directly implementable in an object-oriented programming language. Moreover developers create a sequence diagram for each use case and a state diagram for each class.
- Let us now take a closer look at the relevant parts of the environment that influence the remodelling of the class diagram...



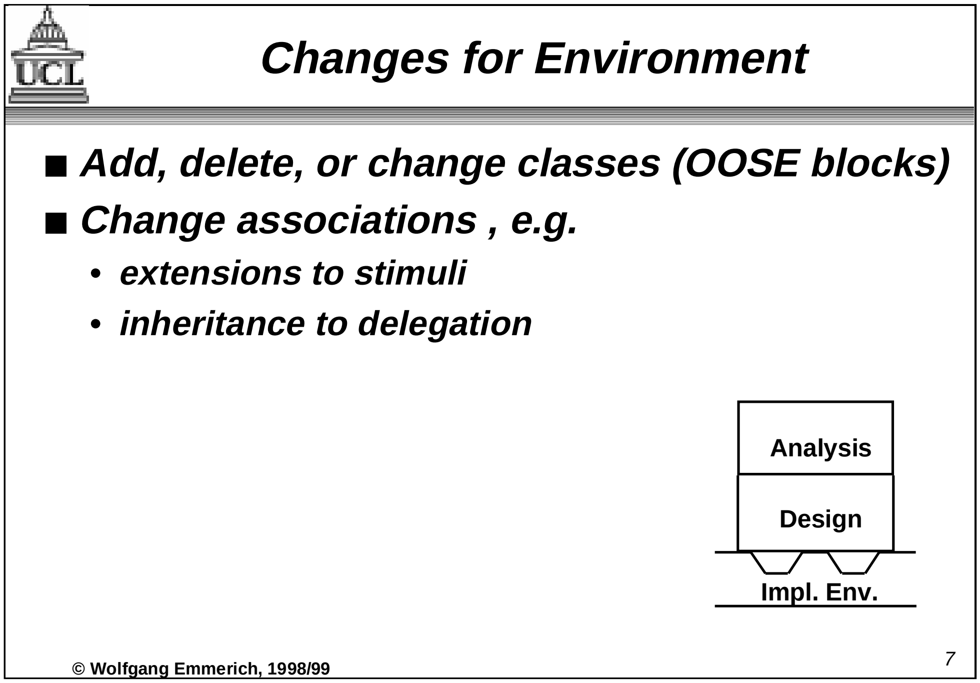
Implementation Environment Factors

- ■ ***Target operating system***
- ■ ***Programming language***
- ■ ***Deployed UIMS***
- ■ ***Available system integration mechanisms***
- ■ ***Underlying DBMS***
- ■ ***Available reuse libraries***
- ■ ***Non-functional requirements***
- ■ ***Development process***

Analysis
Design
Impl. Env.

6

- The target operating system has an influence on the design. Some operating systems (such as the Solaris 2.5 OS installed on CSD machines) support multi-threading, for instance, which means that systems can use concurrent threads and nead not necessarily perform every operation synchronously.Other OS (such as Microsoft's DOS) do not have this capability.
- The programming language used for implementing the design has probably the highest influence on the way the design is used. If the programming language does not support multiple inheritance (e.g. Smalltalk) multiple inheritance that was used in the analysis diagram must be resolved into single inheritance in the design class diagram. Similarly, some programming languages support the redefinition of operation signatures (e.g. Eiffel) while others do not (e.g. C++). In general we must make sure that the design only uses those concepts that are directly mappable to the programming language. Even within different implementations of a programming language there might be differences and the design should only rely on widely available and standardised concepts of the programming language.
- Some of the interface classes are used by humans and a user interface must be constructed. This is typically done by relying on a user interface management system (e.g. X-Windows, OSF/Motif, OpenLook, OpenStep) and the design class diagram must be adapted to the particular UIMS that is in use.
- Other interface classes represent interfaces to existing (legacy) programs and an intergration mechanism, such as OMG/CORBA must be deployed for achieving an integration, this requires the adaption of interface classes to the particular integration mechanism at hand.
- Some objects will have attributes whose values must survive the termination of the system and therefore be stored on persistent storage. Typically database management systems (DBMSs) are used for that purpose. Which particular type of DBMS is used (e.g. an object database or a relational database) has a serious impact on the way the system is designed.
- It might also be the case that existing class libraries (e.g. the Standard Template Library) can be reused and this will simplify the implementation. However, the design then has to identify classes in these libraries and the way in which they are to be deployed.
- Non functional requirements for performance, or restrictions upon it such as memory availability, must be addressed by the design.
- Finally managerial factors can affect the design process, e.g. division of labour between sites, different competences of teams, standard procedures or the decision of a staged deployment.



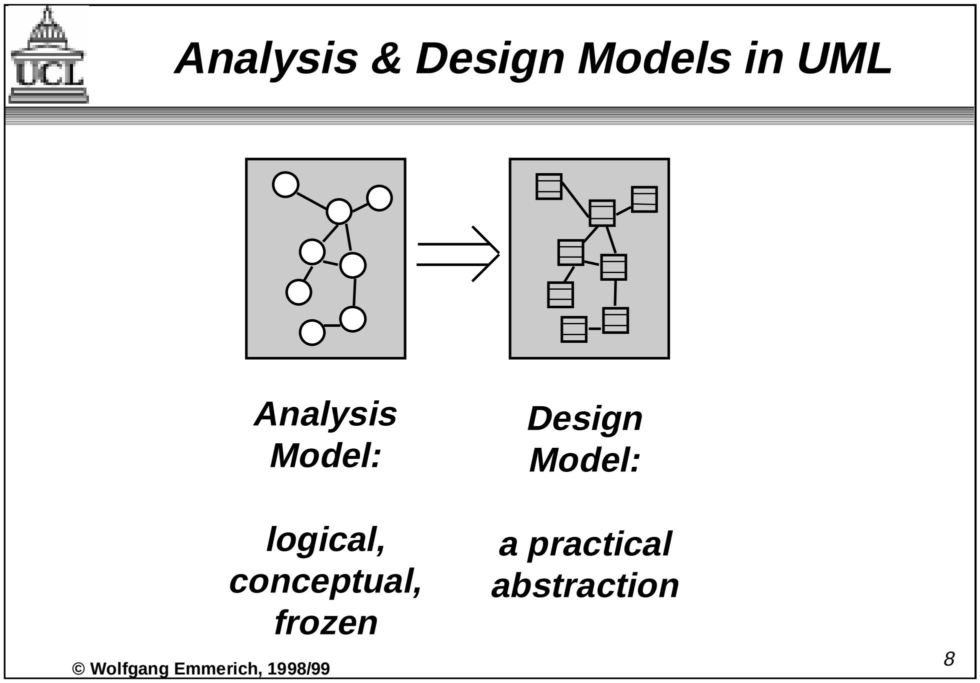
Changes for Environment

- ***Add, delete, or change classes (OOSE blocks)***
- ***Change associations , e.g.***
  - ***extensions to stimuli***
  - ***inheritance to delegation***

Analysis
Design
Impl. Env.

7

- The accomodation of environment specific factors will lead to the addition and/or deletion of classes as well as to classes whose generalisation relationship is changed, that have additional operations and so on.
- Although the class hierarchy might be transformed considerably, the semantics of classes identified during the analysis stage should not be affected. Classes ought just to be notated in a way that is more convenient to implement. Therefore, functional changes, because they imply changes to the analysis model, are suspect. Functional deletions are equally suspect, as are changes that result in splitting or joining blocks for non-environmental reasons.
- Besided accomodating the practicalities of the environment, the change from analysis to design models also involves an important change of perspective as we discuss on the next slide...



Analysis & Design Models in UML

Analysis Model:

logical, conceptual, frozen

Design Model:

a practical abstraction

- - Semantic differences between models are:

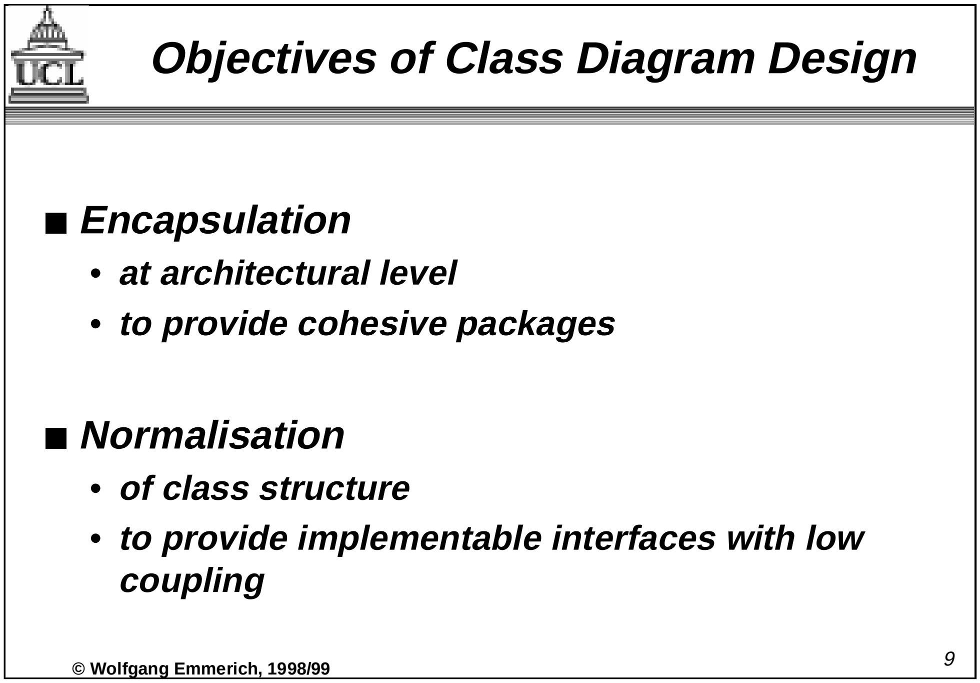
  Analysis model

  - logical model
  - conceptual picture of system
  - frozen at end of analysis process

  Design model

  - abstraction of how system will be built
  - reflecting implementation environment

- Initially there can be a direct translation creating a model which is very similar, particularly having defined ‘entity’ ‘control’ and ‘interface’ classes. However, an important shift has been made to a practical abstraction whilst retaining notation.



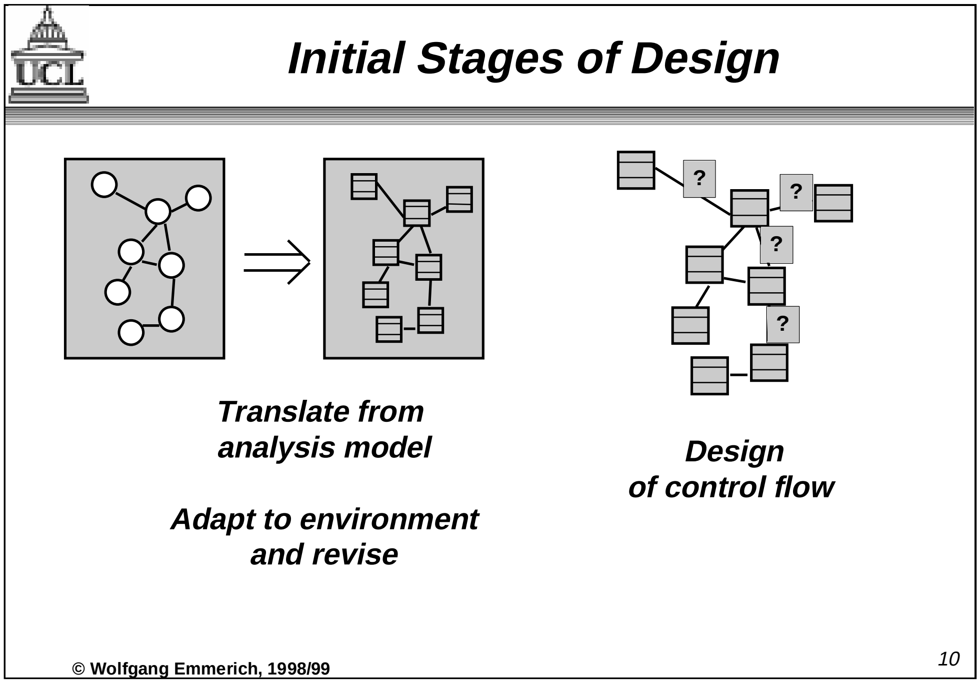
Objectives of Class Diagram Design

- ■ *Encapsulation*
  - *at architectural level*
  - *to provide cohesive packages*
- ■ *Normalisation*
  - *of class structure*
  - *to provide implementable interfaces with low coupling*

© Wolfgang Emmerich, 1998/99

9

- The transfer of the class diagram to a practical abstraction involves activities of ‘encasulation’, as defined earlier but at an architectural level. The aim of achieving encapsulation at an architectural level is to isolate dependencies on external components, such as a UIMS or a DBMS which are likely to change (for instance if the same system has to be deployed for another customer).
- Often packages are used to achieve encapsulation at an architectural level. In the user interface management system example, we would aim at encapsulating the user interface management system in a user interface package that implements all the interface classes for a human/computer interface. If we make sure that not a single definition of the UIMS becomes visible to the outside of that package we only have to adapt the user interface package if we have to port the system to another UIMS.
- Also we aim at designing the classes in a way that its coupling with other classes. The motivation for low coupling derives from the fact that with every additional dependency a class becomes more reliable on its environment and firstly cannot be reused without the other classes it depends on and secondly incremental development becomes more and more difficult because the minimal increment of the system is determined by the transitive closure of the dependency association.
- With normalisation we mean, for instance, that certain naming schemes for attributes and operations are obeyed throughout all classes or that the interfaces of the class are as minimal as possible.



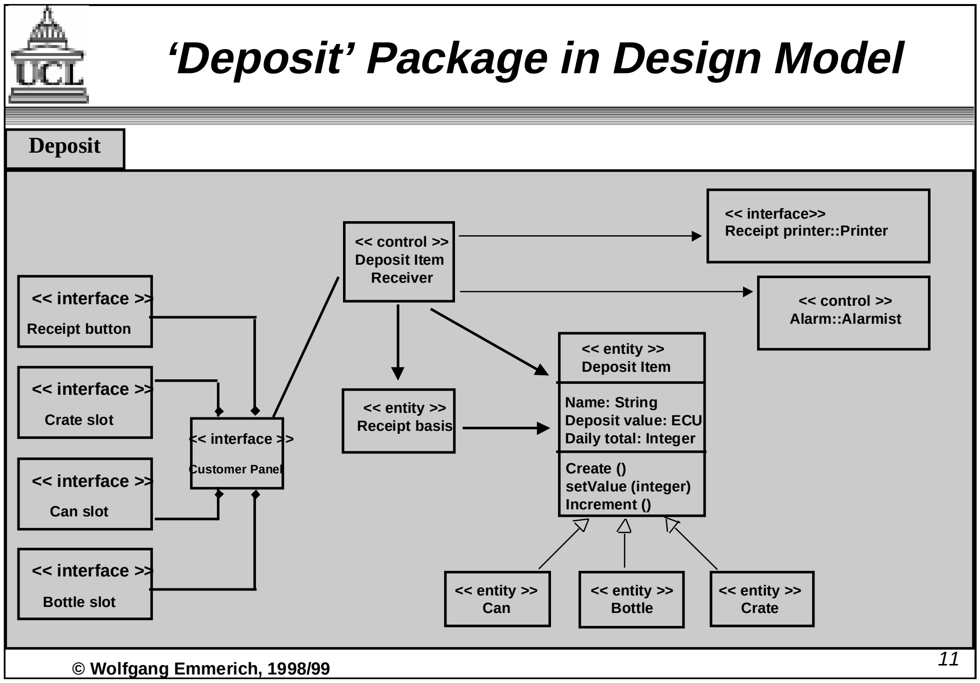
Initial Stages of Design

Translate from analysis model

Adapt to environment and revise

Design of control flow

- Having translated the analysis class model and adapted it to the environment and begun the necessary revisions, there follows a new form of design activity.
- We have convinced ourselves at the analysis stage that the set of classes we have identified are sufficient for all the use cases. In the design stage we undertake a similar activity, though at a more concrete level of abstraction. We will design the control flow between different objects and convince ourselves that
  - we have all the operations that are needed for the implementation of the use cases we have identified
  - that the objects can identify each other based on the associations we have identified.
- To achieve that we are going to model 'sequence diagrams' for each use case. Such a diagram identifies objects that occur within the use case and indicates the temporal order in which messages are exchanged between objects in order to stimulate operation executions.



'Deposit' Package in Design Model

Deposit

<< interface>>
Receipt printer::Printer

<< control >>
Alarm::Alarmist

<< control >>
Deposit Item
Receiver

<< interface >>
Receipt button

<< interface >>
Crate slot

<< interface >>
Customer Panel

<< interface >>
Can slot

<< interface >>
Bottle slot

<< entity >>
Receipt basis

<< entity >>
Deposit Item

Name: String
Deposit value: ECU
Daily total: Integer

Create ()
setValue (integer)
Increment ()

<< entity >>
Can

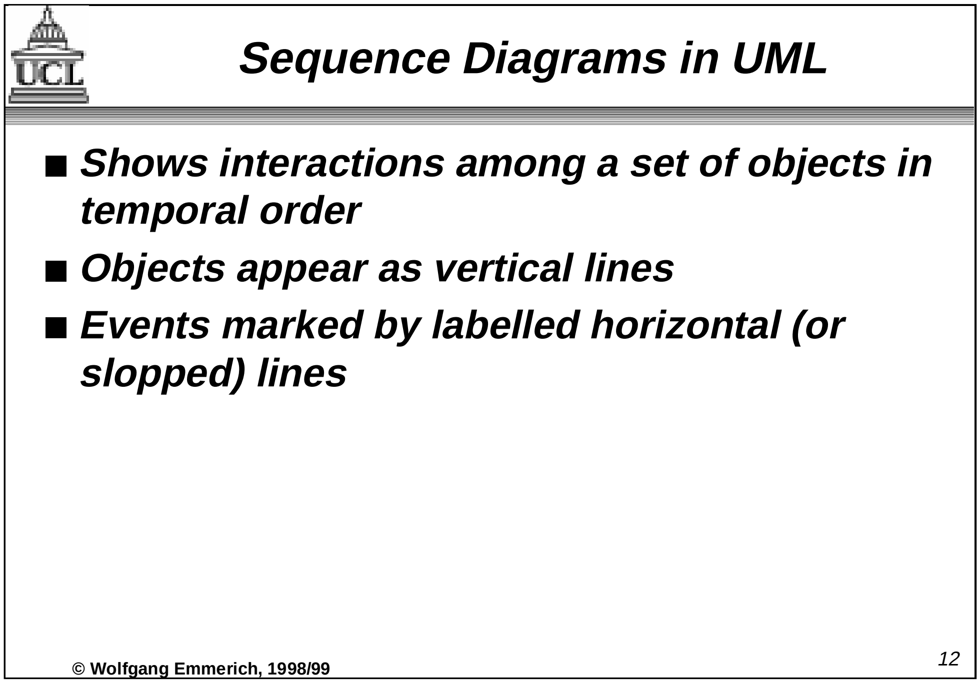
<< entity >>
Bottle

<< entity >>
Crate

© Wolfgang Emmerich, 1998/99

11

- The class diagram so far only indicates associations between classes, their names, direction and multiplicity. We have an initial set of operations derived for entity classes but we are not yet certain whether these operations are sufficient (they almost certainly are not!)
- Moreover, it is still undefined which operation traverses along which association and it is equally undefined which operation uses other operations in order to implement the behaviour associated with it. This is why there are a lot of question marks attached to associations.
- The construction of sequence diagrams provides the essential first step in clarifying the messages passed between all the objects involved. At this point the use cases again take on a central role, by providing views of exactly how parts of the system should interact. These views are modelled in sequence diagrams. The next slide outlines the notation provided in UML for these diagrams.



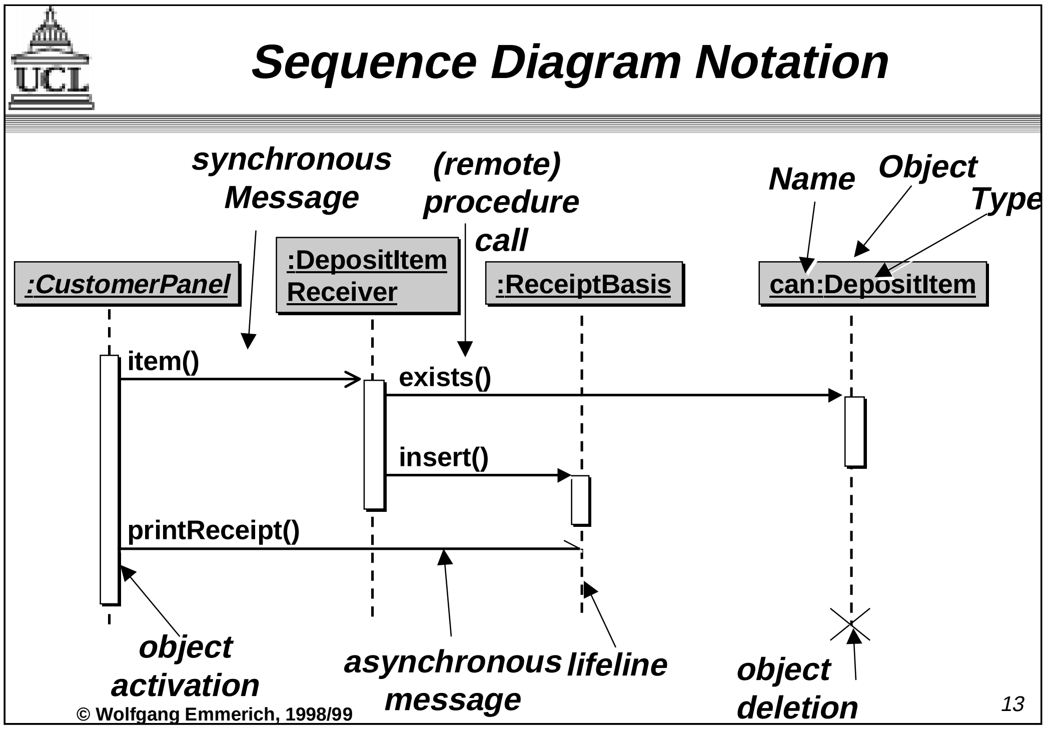
Sequence Diagrams in UML

- ***Shows interactions among a set of objects in temporal order***
- ***Objects appear as vertical lines***
- ***Events marked by labelled horizontal (or slopped) lines***

© Wolfgang Emmerich, 1998/99

12

- A sequence diagram shows a set of objects and the temporal order in which stimuli are sent between them. Stimuli can be sent within (a message that leads to an operation execution) and between processes (a request for a remote operation invocation) and they are not distinguished at this stage.
- Sequence diagrams are a means for showing a ‘scenario’, a particular set of interactions among objects in a single execution of the system. It is essential because the isolated behaviours of individual objects will not give a complete view of a complex system.
- Objects appear as vertical lines. A very thin box (or a broader line) is shown when an object has the thread of control, otherwise the single line represents ‘created but in a waiting state’.
- Events are one-way transmissions of information. They correspond to the OOSE concept of stimuli. Events are marked by a labelled horizontal arrow. Arrows may also slope down (from left to right) when sending and receiving times are distinct (e.g. during a remote CORBA operation request).
- Normally only ‘calls’ to other objects are shown. The returns are implicit (usually when the object ceases to be in the thread of control), but these can be shown explicitly as leftward arrows (for instance if asynchronous communication is assumed). It is also possible to distinguish between ‘in scope’ and ‘in control’ by blocking in sections of the thin boxes.
- Large cross ‘X’ at end of line can show destruction (usually by external command, but in this example we have implicit self-destruction).
- We now return to OOSE to see how sequence diagrams are used...



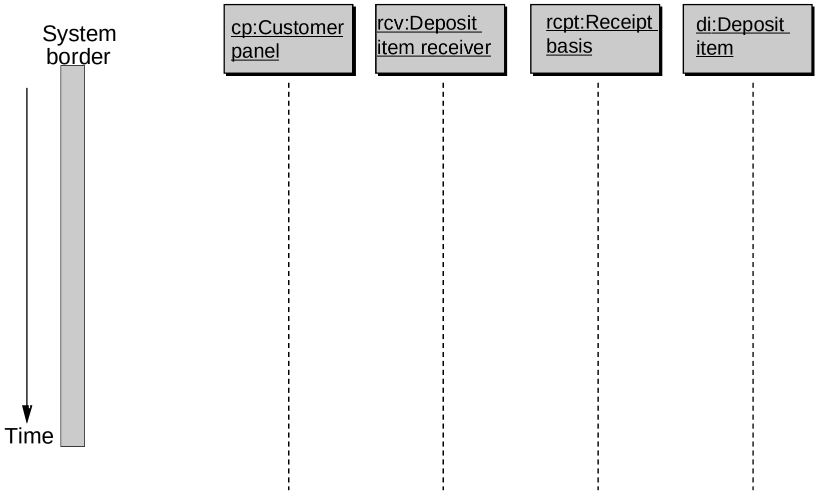
Sequence Diagram Notation

© Wolfgang Emmerich, 1998/99

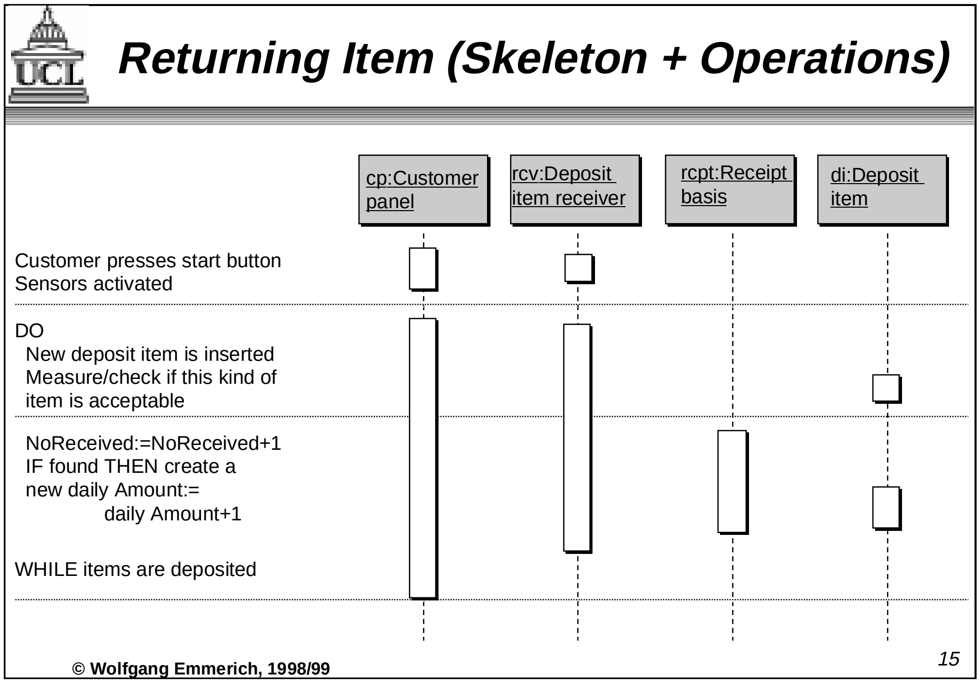
13

- This slide shows the various graphical notations of sequence diagrams.

Returning Item (Skeleton)



- Sequence diagrams are specifications at the instance level. Hence we will have to take archetypical instantiations of use cases, i.e. scenarios, and formalise these as sequence diagrams. This will reveal very useful information that we exploit for completing the design class diagram. Most notably it will identify operations that have not yet been included and that therefore have to be added to the class diagram.
- We do so by first identifying the objects (i.e instances of classes identified in the analysis class diagram) that are involved in the scenario and draw them at the top edge of the diagram. The system border identifies stimuli that come from outside the system (e.g. a user or another system). We will omit that border later.
- This slide displays merely the ‘skeleton’ of a scenario derived from the returning item use case that we have used as a running example. The sequence diagram will be continued on the next slide...

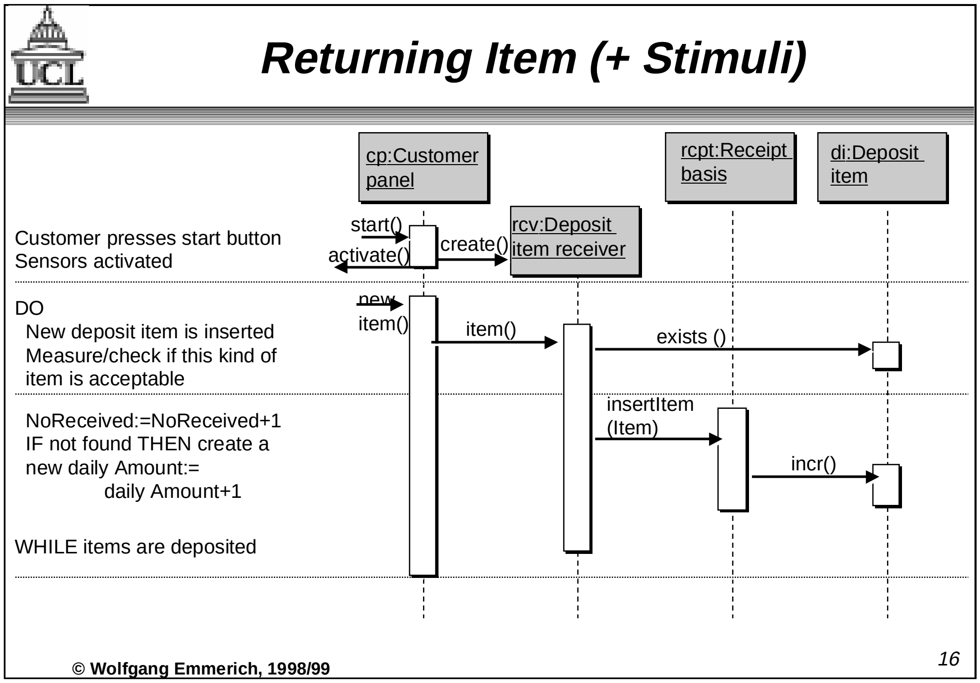


Returning Item (Skeleton + Operations)

```
Customer presses start button
Sensors activated

DO
  New deposit item is inserted
  Measure/check if this kind of
  item is acceptable

  NoReceived:=NoReceived+1
  IF found THEN create a
  new daily Amount:=
            daily Amount+1

WHILE items are deposited
```

15

- The next stage is the identification of an algorithm that describes how the scenario is performed. The vertical boxes are used to distribute the tasks identified in the algorithmic specification over objects. Length and vertical position of boxes may not be exact at this point.
- The text on the left is a ‘pseudo code’ version of the operations involved in the use case. The generation of this text is part of what Jacobson terms ‘use case design’, i.e. the formalisation of the use case as a step towards implementable code. This text is not part of the diagram defined in UML, but provides a useful and non-contradictary supplement.
- Next we need to identify how the different objects communicate. This is based on stimuli...



Returning Item (+ Stimuli)

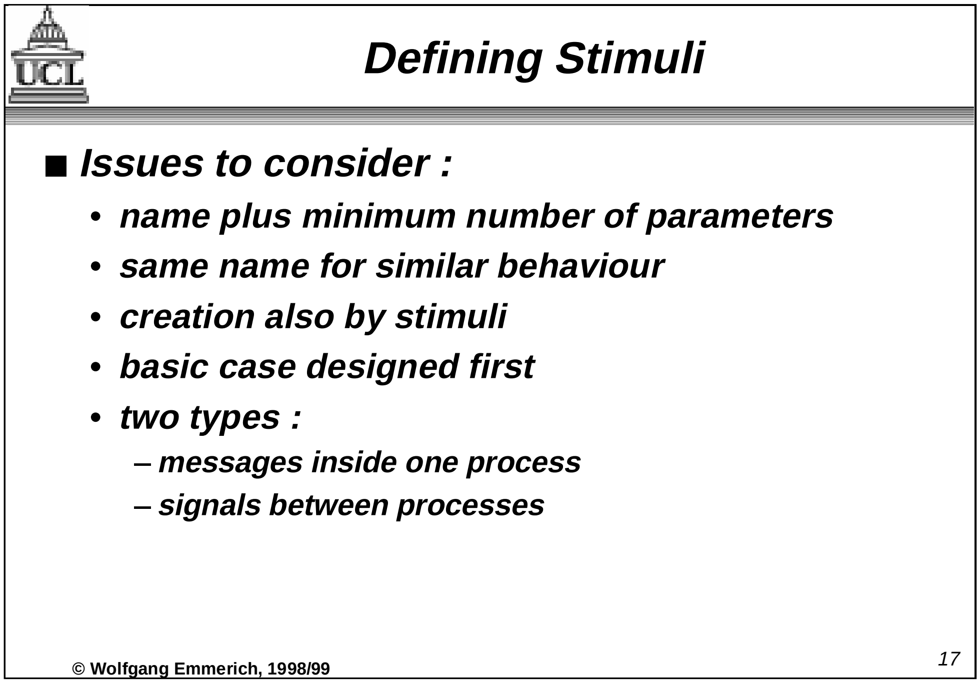
Customer presses start button
Sensors activated

DO
New deposit item is inserted
Measure/check if this kind of item is acceptable

NoReceived:=NoReceived+1
IF not found THEN create a new daily Amount:=
daily Amount+1

WHILE items are deposited

- This slide displays a more complete version of a 'Returning Item' scenario that also shows the timing of events.
- We can see that the scenario is activated by a 'start()' event that is released by the customer pressing the start button. It leads to the execution of the 'start()' operation in the customer panel object 'cp'. That operation creates a deposit item receiver object 'rcv' and activates the external sensors. It then waits for items to be inserted.
- A 'newitem()' event occurs from the outside whenever a customer inserts a new deposit item into the recycling machine. This event leads to the execution of operation 'newitem()' in 'cp'. The panel 'cp' then delegates the execution to the deposit item receiver object 'rcv'. It checks whether the item inserted is a proper recycling item by invoking operation exists from the deposit item object 'di'. If the item is known as a recyclable object, the item is inserted into the receipt basis object 'rcpt' to make sure that a description of the object is included if the customer wishes to obtain a receipt. The length of the list of received items internally managed by 'rcpt' will represent the 'noReceived'. Finally the daily amount of items returned for a particular class of deposit items is incremented.
- Next, we need to consider a number of issues in defining stimuli...

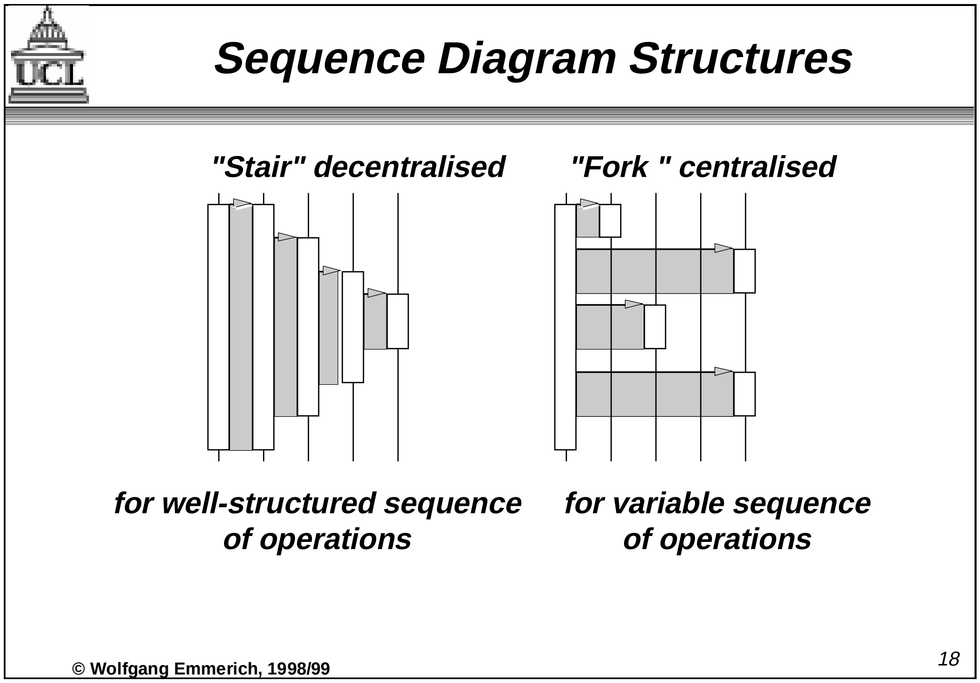


Defining Stimuli

- **_Issues to consider :_**
  - **_name plus minimum number of parameters_**
  - **_same name for similar behaviour_**
  - **_creation also by stimuli_**
  - **_basic case designed first_**
  - **_two types :_**
    - **_messages inside one process_**
    - **_signals between processes_**

17

- The naming of events (aka oprations) should be done with great care. The ease of understanding of these sequence diagrams and the later maintenance of the system very much depend on the fact that names give the reader a clear idea as to what events/operations mean. One should try to establish and obey naming conventions and, for instance, use the same name for similar behaviour in different classes.
- Also parameter lists should be kept as small as possible. Lengthy parameter lists are an indication that the operation is not really atomic but rather performs many different tasks. Then the operation should be split up into more simple operations. Simpler operations are easier to use and the probability that they can be reused increases.
- All naming issues are associated with the needs for understanding and reusability. Similar considerations apply to the requirement to minimise the number of parameters asociated with particular messages.
- Creation of objects is the result of specific events. The creation of an object in an object-oriented programming language is done by sending a message to a class (rather than an object). The class will execute a special creation operation (called constructor), perform the initialisations specified in the constructor and return the identification of the newly created object as a result.
- Modelling of sequence diagrams should start with a scenario that reflects the basic course of events in the use case. When that has fully been understood will the designer be in a position to model more special and exceptional scenarios.



Sequence Diagram Structures

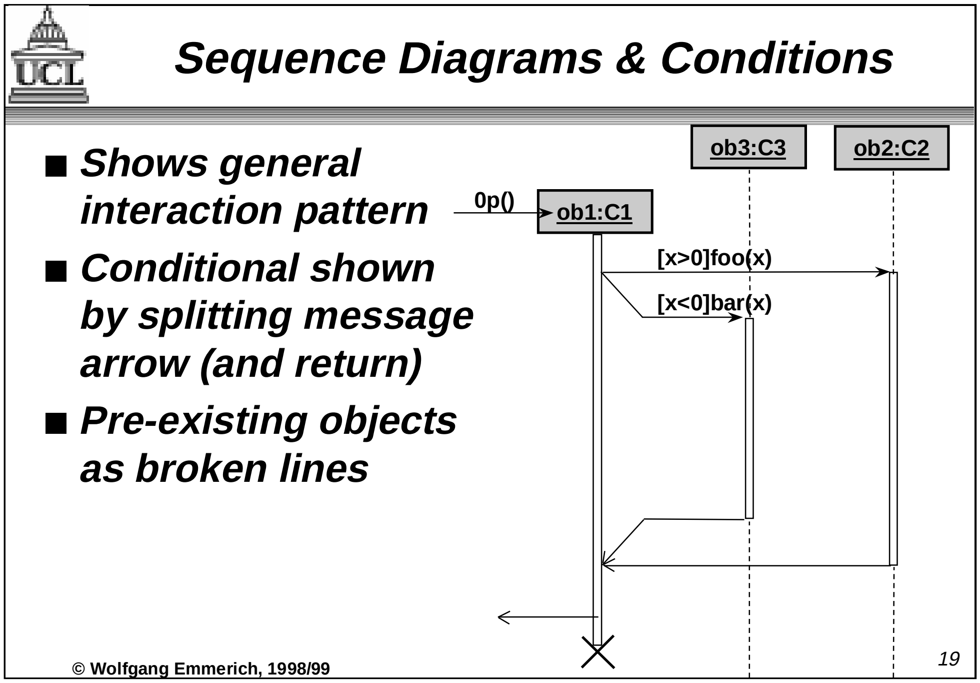
"Stair" decentralised

"Fork " centralised

for well-structured sequence of operations

for variable sequence of operations

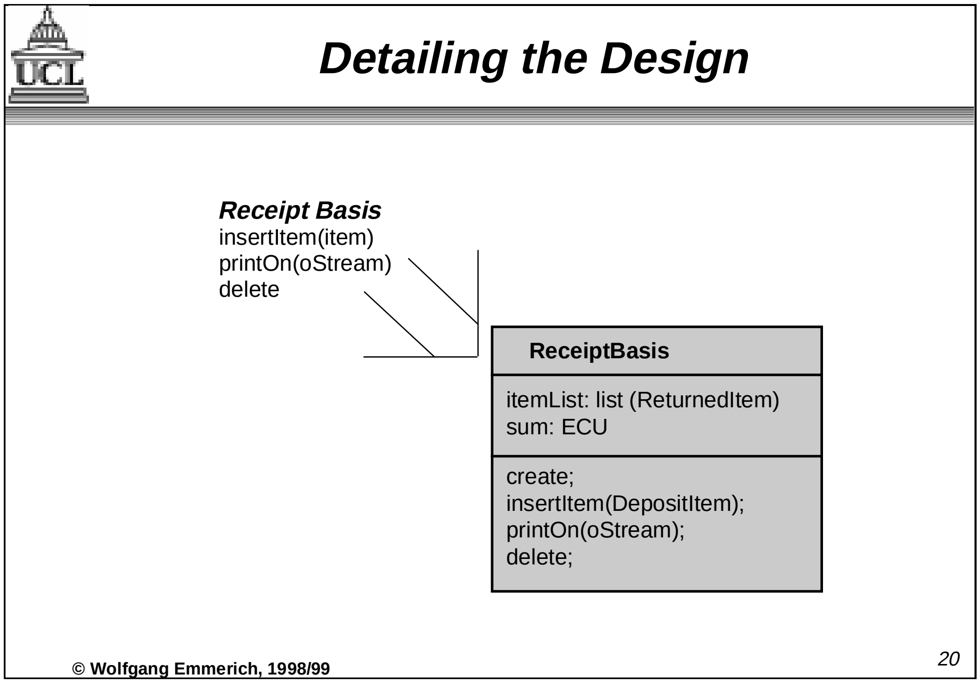
- The sequence diagram on the left-hand side follows a decentralised control pattern. It is decentralised, because the depth of the call stack is considerable (five in this example). This implies that many operations are involved in the thread of control.
- The right-hand side sequence diagram follows a centralised control pattern as the object displayed on the left of that diagram keeps full control and the objects that it stimulates return control immediately rather than stimulating other objects.
- Decentralised contol is appropriate if operations have a strong connection with hierarchical or fixed temporal relationships (as in ‘Returning item’). Centralised pattern is more suitable if operations can change order; new operations can be inserted.
- Next, we review how control flow directives can be included in sequence diagrams...



Sequence Diagrams & Conditions

- Shows general interaction pattern
- Conditional shown by splitting message arrow (and return)
- Pre-existing objects as broken lines

- A scenario is an instantiation of a use case and as such it only has a single flow of events. Conditions and loops have been checked beforehand and are therefore not reflected in the diagram. Although this keeps the diagram simple, it is not as expressive as it could be and many sequence diagrams are needed to provide a full cover of the possible sequence of events in a use case.
- In order to increase the expressiveness (and at the same time reduce the number of sequence diagrams needed), UML includes facilities for expressing control structures, such as conditions and loops.
- The slide above displays the UML notation for a condition. The guard condition appears in square brackets. The expression must be unambiguous (In the example X=0 not included and therefore no branch in this case)
- As you have seen now, sequence diagrams provide an essential means for elaborating which operations are needed and how they cooperate towards the implementation of the system (as required in different use cases).



Detailing the Design

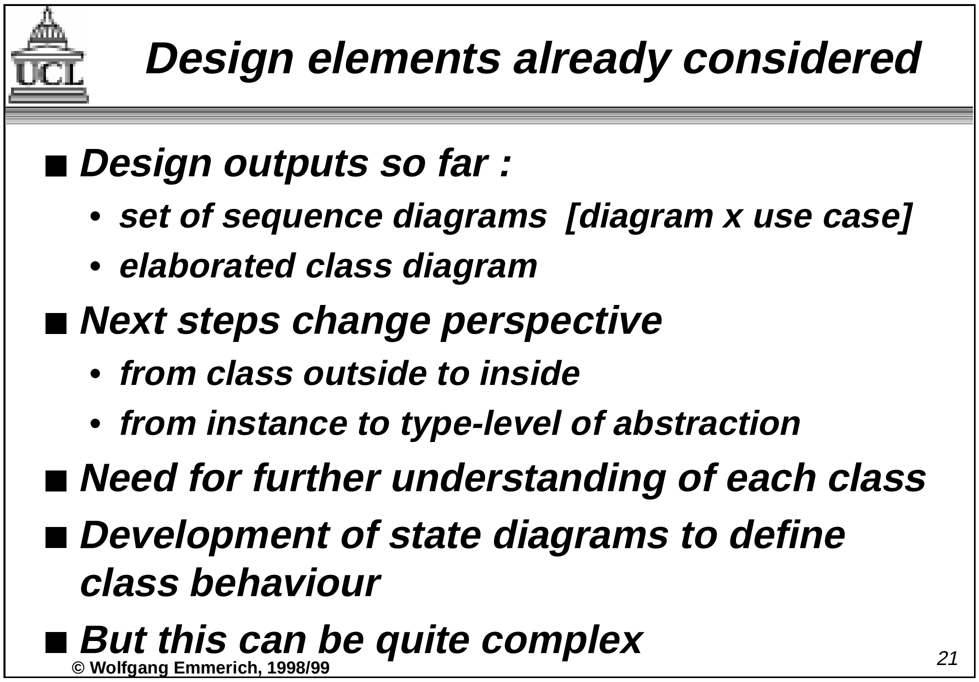
Receipt Basis
insertItem(item)
printOn(oStream)
delete

ReceiptBasis

itemList: list (ReturnedItem)
sum: ECU

create;
insertItem(DepositItem);
printOn(oStream);
delete;

- Next we can exploit the sequence diagrams in order to complete the class diagrams. In particular we use the information about events and their parameters in order to devise operations and their signatures in the class diagram.
- We can also use it to define the public and the private parts of each class. Every operation that needs to be executed in response to an event sent by another object has to be public. We can freely add private operations in order to use them in the implementation of the public operations. We then have completed the first interface design.
- Jacobson suggests to start the implementation of classes identified in the design class diagram only when the class interfaces (i.e. the public operations) begin to stabilise. This means in particular, that it is not necessary to fully design the whole class diagram before the first classes are coded.
- <u>But</u> it might still be necessary to perform further work on the design class diagram, in particular on the characteristics of individual classes, rather than use cases.



Design elements already considered

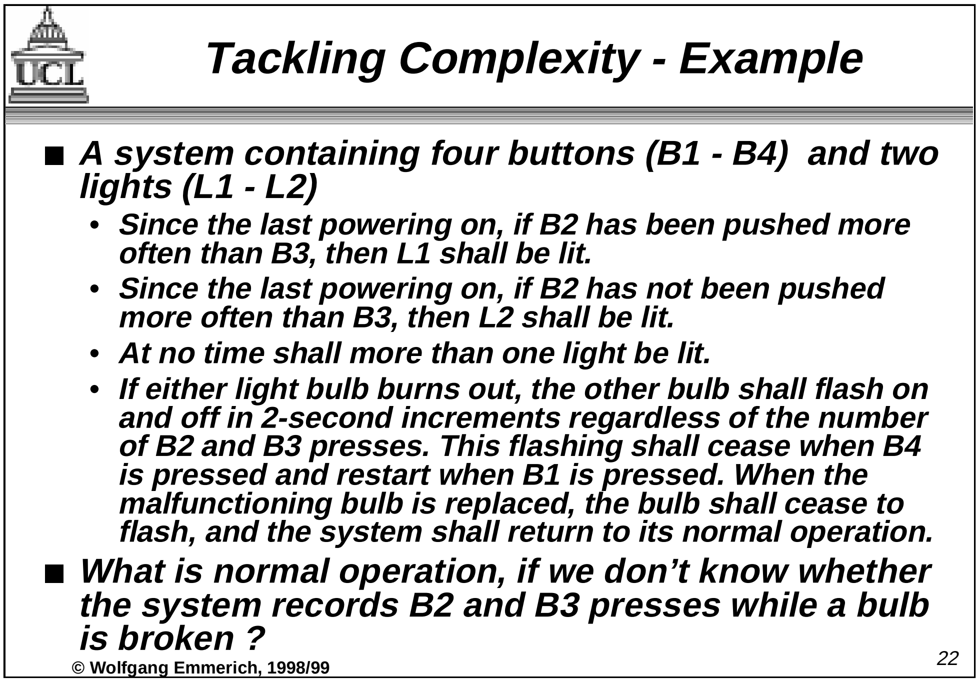
- ■ ***Design outputs so far :***
  - ***set of sequence diagrams [diagram x use case]***
  - ***elaborated class diagram***
- ■ ***Next steps change perspective***
  - ***from class outside to inside***
  - ***from instance to type-level of abstraction***
- ■ ***Need for further understanding of each class***
- ■ ***Development of state diagrams to define class behaviour***
- ■ ***But this can be quite complex***

- The elements of the design that we have considered already are :

  a) the set of sequence diagrams, each representing the temporal interaction of all the objects in a single scenario, i.e an instance of a particular use case, and,

  b) the elaborations made to the class diagram, in the form of operations derived from the sequenced diagrams and incorporated into the interface of each class.

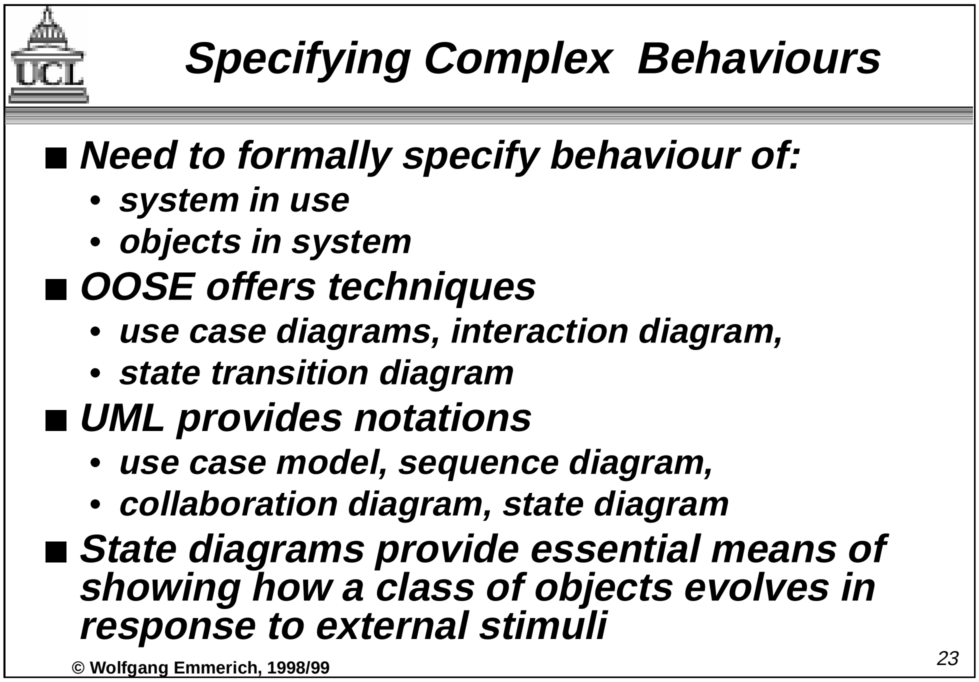
- In preparing the next steps, we need to consider both the ‘system in use’ (as represented in the various sequence diagrams) and the ‘objects in the system’ and how each will evolve in response to external stimuli.
- State diagrams (the subject of the this lecture) provide the essential means of describing the dynamic behaviour of a class, via the temporal evolution of an object in response to interactions with other objects inside or outside the system.
- The state diagrams are a mathematical well defined language. They are based on the concept of finite state machines. Hence, we are going to introduce this concept first...



Tackling Complexity - Example

- ***A system containing four buttons (B1 - B4) and two lights (L1 - L2)***
  - ***Since the last powering on, if B2 has been pushed more often than B3, then L1 shall be lit.***
  - ***Since the last powering on, if B2 has not been pushed more often than B3, then L2 shall be lit.***
  - ***At no time shall more than one light be lit.***
  - ***If either light bulb burns out, the other bulb shall flash on and off in 2-second increments regardless of the number of B2 and B3 presses. This flashing shall cease when B4 is pressed and restart when B1 is pressed. When the malfunctioning bulb is replaced, the bulb shall cease to flash, and the system shall return to its normal operation.***
- ***What is normal operation, if we don’t know whether the system records B2 and B3 presses while a bulb is broken ?***

- This example and other material for this lecture have been taken from [Davis90].
- The problem is that with the specification techniques that we have seen so far we cannot unambiguously specify the behaviour of the system. Scenarios are not suitable as we might have to include a very high number of scenarios in order to completely describe the system.
- Such questions often arise in dynamic or reactive systems in which input data continues to arrive during processing to effect the program’s outcome, so-called ‘real-time’ systems.
- Moreover, use cases are described in natural language (usually English in this country) and the use of a natural language often leaves room for different interpretations. This is particularly inapropropriate in situations where the behaviour of the system must be specified very precisely (think of the bulbs and buttons as part of an aircraft control panel). In these, so called safety-critical systems, imprecisely specified behaviour risks human lives.
- What means are their for specifying and representing behaviour precisely?



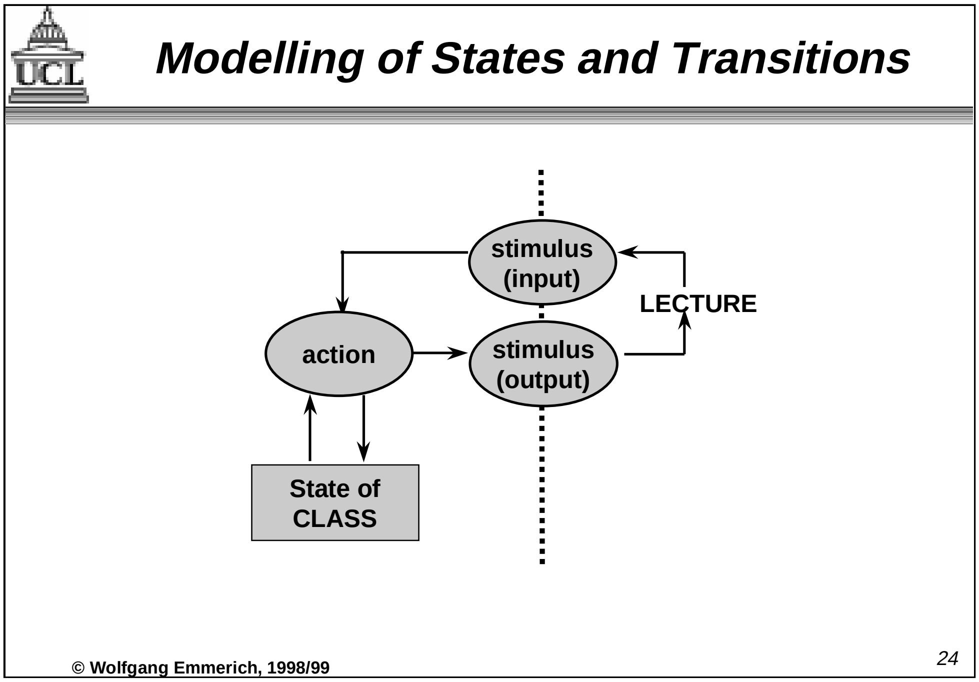
Specifying Complex Behaviours

- ■ ***Need to formally specify behaviour of:***
  - ***system in use***
  - ***objects in system***
- ■ ***OOSE offers techniques***
  - ***use case diagrams, interaction diagram,***
  - ***state transition diagram***
- ■ ***UML provides notations***
  - ***use case model, sequence diagram,***
  - ***collaboration diagram, state diagram***
- ■ ***State diagrams provide essential means of showing how a class of objects evolves in response to external stimuli***

© Wolfgang Emmerich, 1998/99

23

- Both use cases and sequence diagrams (called interaction diagrams in OOSE) are concerned with the system in use. They display the flow of messages according to different scenarios, potentially involving a number of objects.
- UML also provides a ‘collaboration diagram’ which shows interaction between a set of objects as nodes in a graph, thus emphasizing relationships rather than temporal flow of behaviour shown in the sequence diagram. This has not been included because firstly it has no counterpart in OOSE and secondly the purpose of collaboration diagrams can equally well be met through sequence diagrams.
- State diagrams come in various forms and provide powerful tools for the design of the behaviour in complex systems.
- The concept of the ‘finite state machine’ underlies all state diagrams and provides the theoretical means of defining the state of a system and its reactions to new stimuli.



Modelling of States and Transitions

stimulus (input)

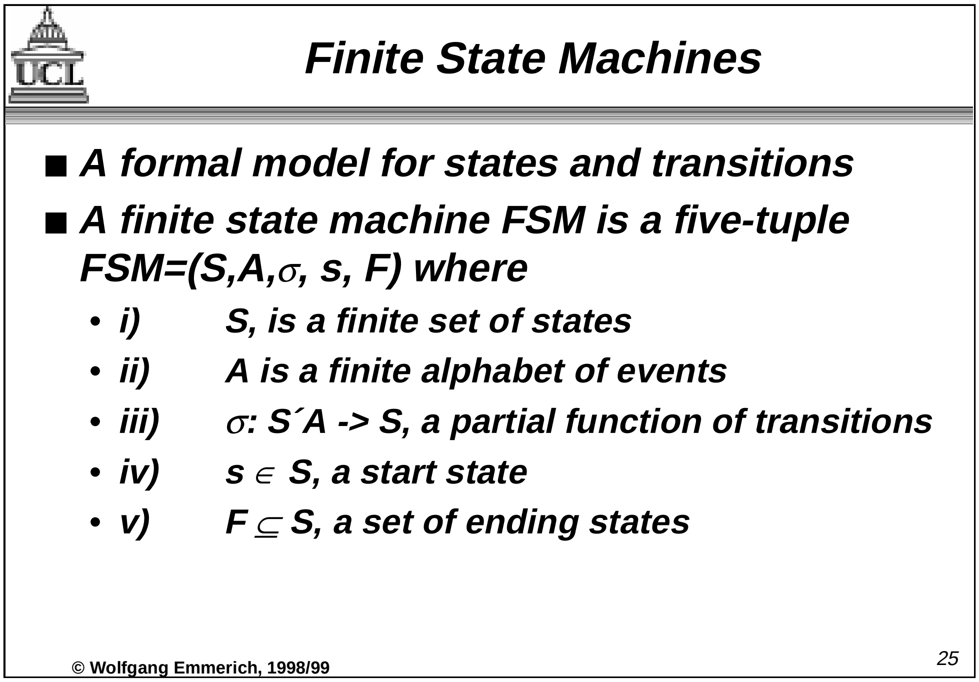
LECTURE

action

stimulus (output)

State of CLASS

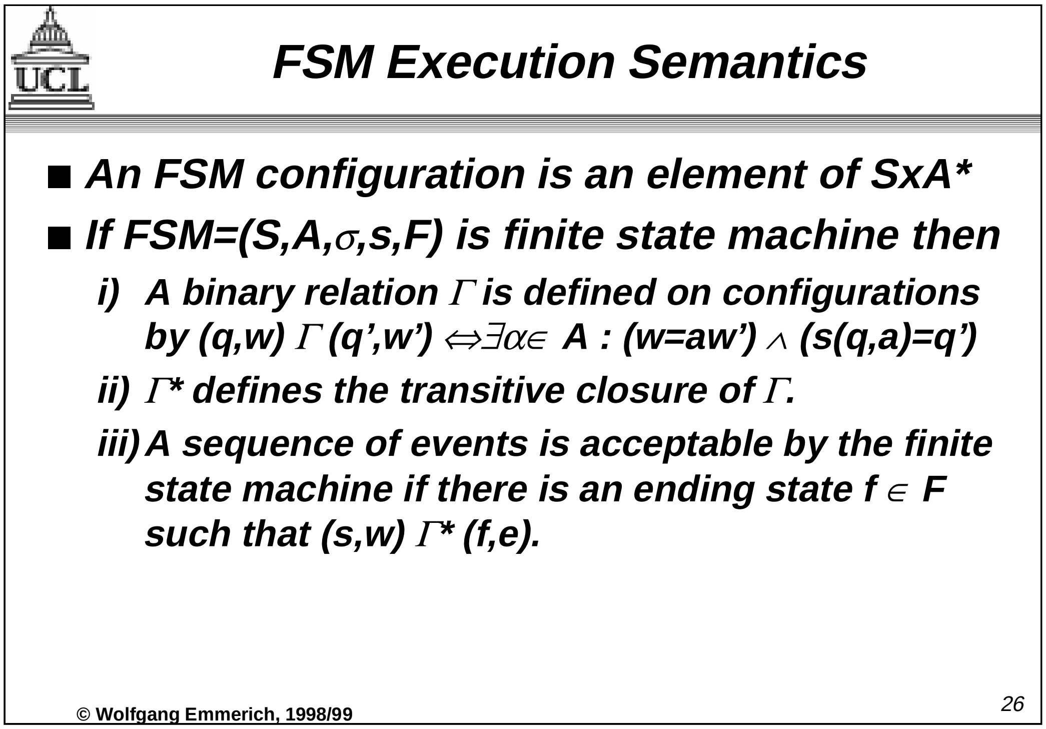
- A finite state machine is a hypthetical machine which allows, for example, the modelling of the behaviour of this class in the context of this lecture.
- The lecturer provides a series of inputs, causing actions of some kind of action in the class, which then generates an output and, if the lecture is effective, causes a permanent change in its state.
- Let us know consider the mathematical definition of finite state machines...



Finite State Machines

- ***A formal model for states and transitions***
- ***A finite state machine FSM is a five-tuple FSM=(S,A,σ, s, F) where***
  - ***i) S, is a finite set of states***
  - ***ii) A is a finite alphabet of events***
  - ***iii) σ: S´A -> S, a partial function of transitions***
  - ***iv) s $\in$ S, a start state***
  - ***v) F $\subseteq$ S, a set of ending states***

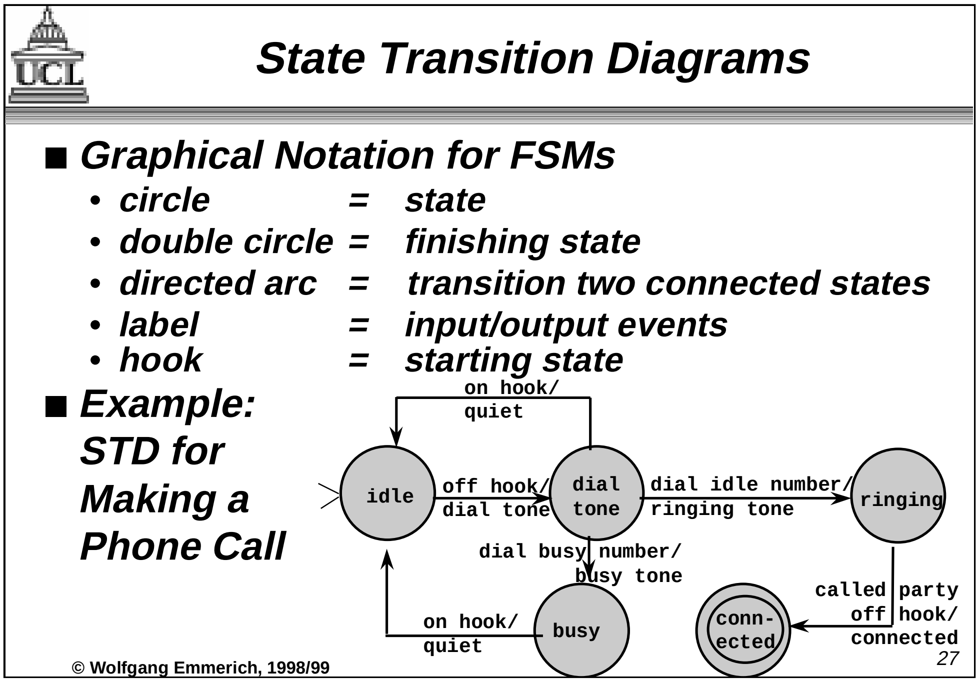
- A finite state machine (FSM) includes a finite set of states (S) one particular element of that set is a starting state (s) and a subset F of S is designated as the set of ending states. Finite state machines in general work on alphabets of characters. For the purpose of this lecture, we can consider these alphabets to denote a finite set of events. The core of any FSM then is the transition function that defines for a pair consisting of a state and an event the successor state.
- For the defnition of the semantics of an FSM machine, we need the concept of a configuration, which denotes the current state and a sequence of events that remain to be processed.



FSM Execution Semantics

- An FSM configuration is an element of SxA*
- If FSM=(S,A,σ,s,F) is finite state machine then
  - i) A binary relation Γ is defined on configurations by (q,w) Γ (q’,w’) $\Leftrightarrow \exists \alpha \in$ A : (w=aw’) $\wedge$ (s(q,a)=q’)
  - ii) Γ* defines the transitive closure of Γ.
  - iii) A sequence of events is acceptable by the finite state machine if there is an ending state f $\in$ F such that (s,w) Γ* (f,e).

- The relation Γ gives the semantics to the FSM. It defines a single step of execution. If the FSM is currently in state q and event α is the next event that is to be processed the new state will be q' if σ(q,α)=q'. The transitive closure of Γ denotes the set of reachable states and a sequence of events is acceptable only if the state the application of the sequence of events to the start state leads to an ending state.
- This notation for finite state machines is mathematically sound. However, it is not an appropriate notation for humans to understand these machines. We will now use state transition diagrams and state transition matrices as notations whose semantics is formally defined based on finite state machines...



State Transition Diagrams

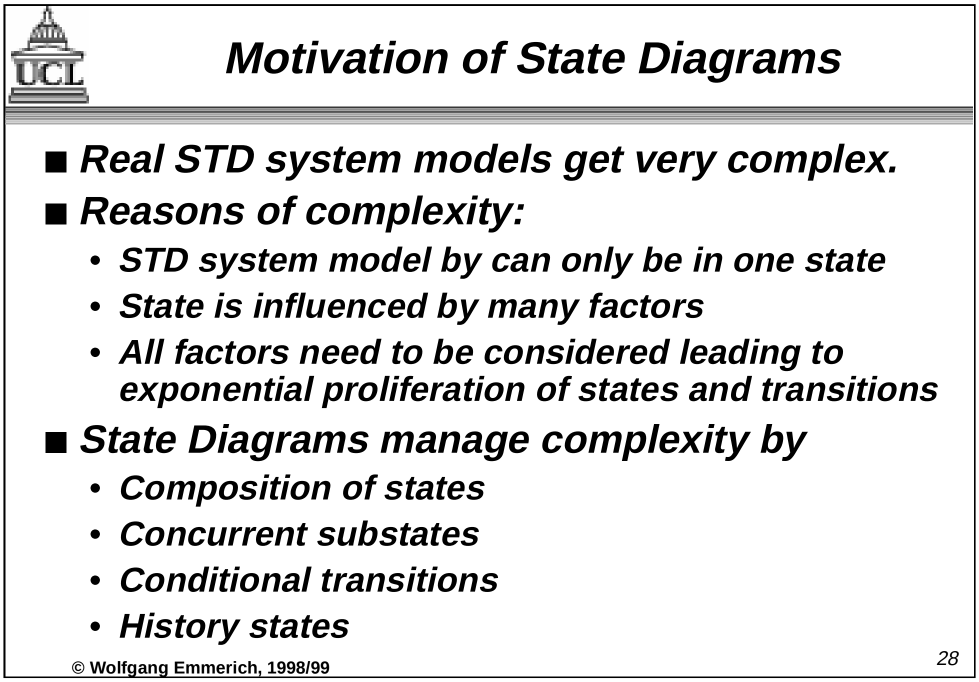
- ***Graphical Notation for FSMs***
  - ***circle = state***
  - ***double circle = finishing state***
  - ***directed arc = transition two connected states***
  - ***label = input/output events***
  - ***hook = starting state***
- ***Example: STD for Making a Phone Call***

idle
dial tone
ringing
busy
conn-ected
off hook/ dial tone
on hook/ quiet
dial idle number/ ringing tone
dial busy number/ busy tone
called party off hook/ connected
on hook/ quiet

© Wolfgang Emmerich, 1998/99

27

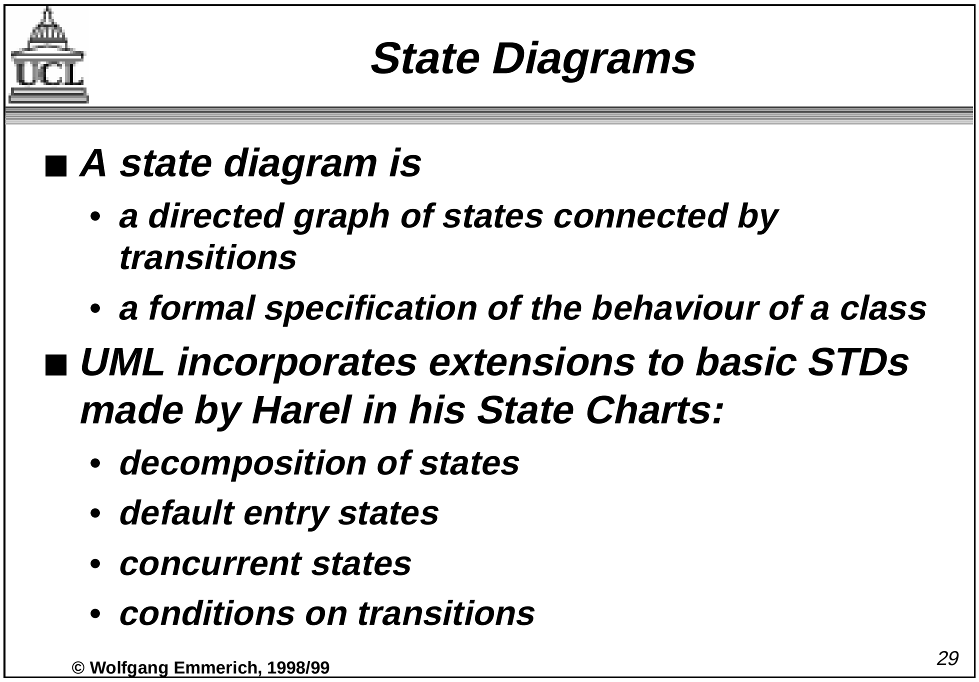
- State transition diagrams are a direct representation of finite state machines. States are represented as circles and labelled arrows between states denote that the function σ is defined for the state where the arrow starts and produces the state where the arrow leads to if the event occurs that is recognised by the label.
- All ending states are denoted as a double circle and the starting state is denoted as a circle where an open arrow head leads to (idle in the example above). Note, that there are also special forms of finite state machines that produce an output whenever a transition occurs. Outputs, if any, are given after the event definition and the two are separated by a slash.
- The example displays states for a telephone. Initially, the telephone is in an idle state. If the receiver is taken off the hook a dial tone will be played. If the receiver is replaced the telephone will become idle again. If a number is dialled of a phone that is busy, the busy tone will be played and the only possible event is to replace the receiver onto the hook. If an idle number is dialed the ringing tone will be played and as soon as the other party takes the receiver off the hook the phone connection will be established.
- An alternative notation to state transition diagrams are state transition matrices...



Motivation of State Diagrams

- ***Real STD system models get very complex.***
- ***Reasons of complexity:***
  - ***STD system model by can only be in one state***
  - ***State is influenced by many factors***
  - ***All factors need to be considered leading to exponential proliferation of states and transitions***
- ***State Diagrams manage complexity by***
  - ***Composition of states***
  - ***Concurrent substates***
  - ***Conditional transitions***
  - ***History states***

- If any realistic system is modelled with state transition diagrams these diagrams get **very** complex. The reason for this complexity is that finite state machines have exactly one active state and that state has to be explicitly modelled in the respective state transition diagram.
- In reality, however, states can be influenced by a number of different factors. A telephone receiver can be either idle or it can be active. If it is active, it can be playing a busy tone, a ringing tone or a dial tone. State transition diagrams do not properly support the different levels of abstractions involved in this example.
- This leads to an exponential growth in the number of states and transitions needed and makes the resulting diagrams unmanageable.
- David Harel had to formally define the behaviour of a fighter aircraft component (on behalf of the Isralian air force). He found that engineers and pilots could easily understand state transition diagrams and searched for a way to cope with their complexity rather than introducing a completely different formalism nobody would be familiar with. The approach he took is, in fact, based on the same that we suggested in the first lecture: abstraction.
- In the state charts that he suggested, he introduced facilities for considering states at different levels of abstraction. He then introduced different notions for composing abstract states from more concrete states which might have internal state transitions embedded. These internal state transitions, however, would be hidden at the more abstract level; hence he applied the principle of information hiding also to modelling of states.
- The UML includes state diagrams as a notation for state charts. State diagrams will be used to model the behaviour of objects and in particular the state transitions of objects in response to events, internal or external to the

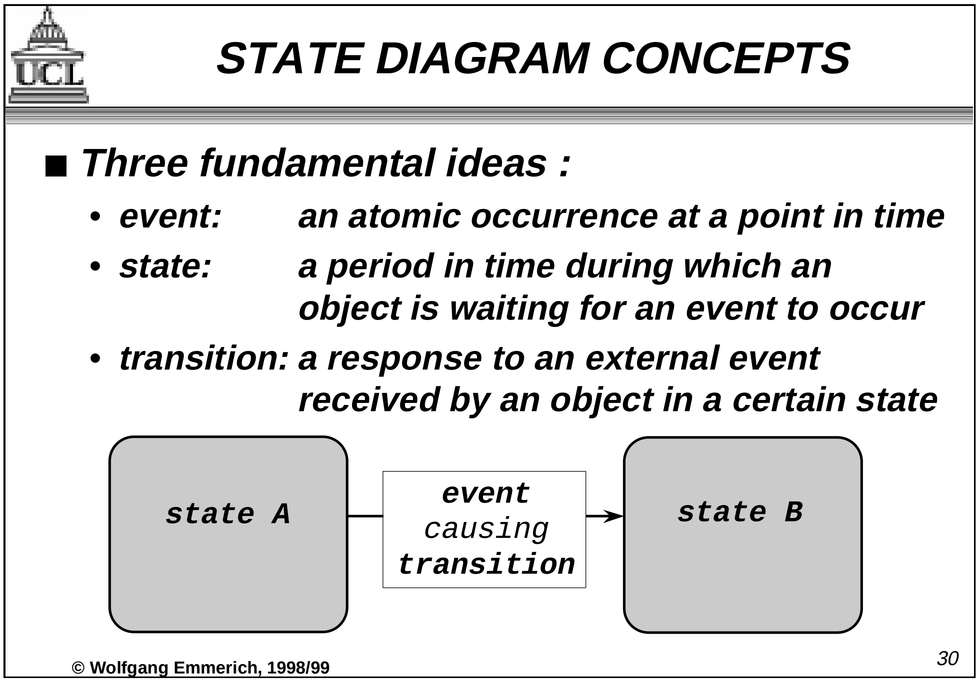


State Diagrams

- **_A state diagram is_**
  - **_a directed graph of states connected by transitions_**
  - **_a formal specification of the behaviour of a class_**
- **_UML incorporates extensions to basic STDs made by Harel in his State Charts:_**
  - **_decomposition of states_**
  - **_default entry states_**
  - **_concurrent states_**
  - **_conditions on transitions_**

29

- The representation of state charts in state diagrams takes the form of a collection of nodes (the states) and directed edges (the transitions).
- A ‘scenario’, being an instance of a use case and an instance of the execution of a system, illustrates, but ultimately cannot define, behaviour. It is a ‘slice’ of system behaviour across state diagrams from multiple classes. The state diagram provides the means for describing the temporal evolutions of an object of a given class in response to interactions with other objects. Hence, the state diagram subsumes the different sequence diagrams that model scenarios from an object-level perspective.
- Each diagram is associated with one class, or with some higher level state. Only a minority of classes undergo significant state changes necessitating diagrams.
- The UML documention acknowledges the contribution of Harel. His important extensions to state transition diagrams provide a useful basis for examining the main elements of state diagram which can be modelled using the UML.



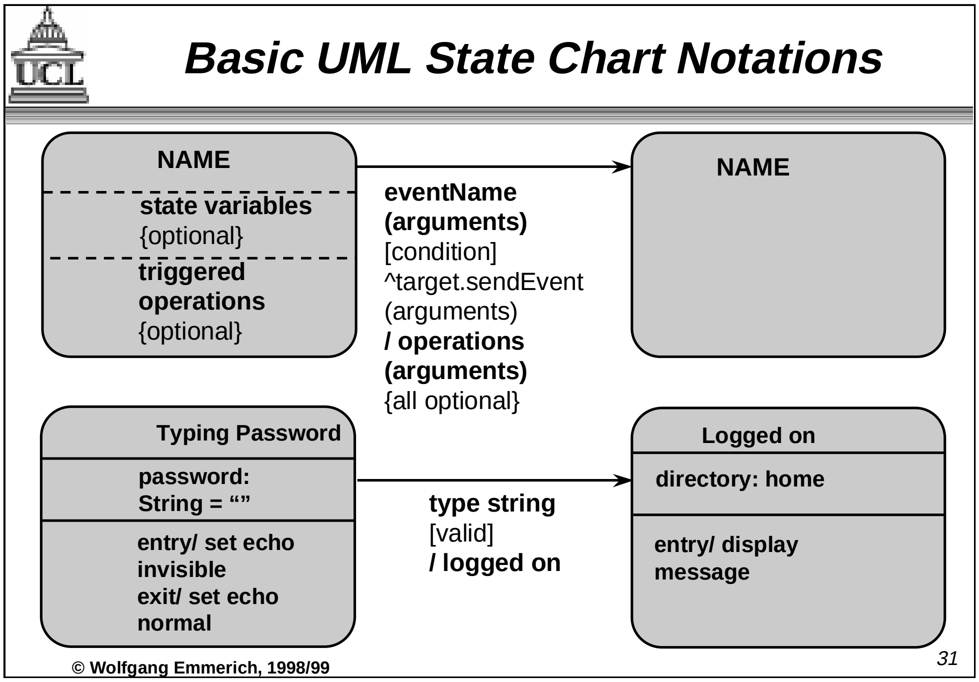
STATE DIAGRAM CONCEPTS

- ***Three fundamental ideas :***
  - ***event:*** ***an atomic occurrence at a point in time***
  - ***state:*** ***a period in time during which an object is waiting for an event to occur***
  - ***transition:*** ***a response to an external event received by an object in a certain state***

state A
event
causing
transition
state B

30

- The basic concepts of state charts are applicable at all levels of abstraction, althouth state diagrams in UML (and the related notation in OOSE) are principally intended to describe the behaviour of objects at a type-level of abstraction, i.e. in classes (or design blocks in OOSE).
- In defining an event, the emphasis is on its atomicity; it is a non-interruptible, one-way transmission of information from one object to another, proceeding independently (asynchronous).
- The current state of an object is determined by the event that triggered its last transition and lasts until the next significant event.
- A transition may both a) cause a change of state and b) invoke object operations. External transitions do a) and possibly b); internal transitions do b) but not a).
- UML provides a multi-featured graphical representation for these concepts as detailed on the next slide...



Basic UML State Chart Notations

NAME

state variables
{optional}

triggered
operations
{optional}

eventName
(arguments)
[condition]
^target.sendEvent
(arguments)
/ operations
(arguments)
{all optional}

NAME

Typing Password

password:
String = “”

entry/ set echo
invisible
exit/ set echo
normal

type string
[valid]
/ logged on

Logged on

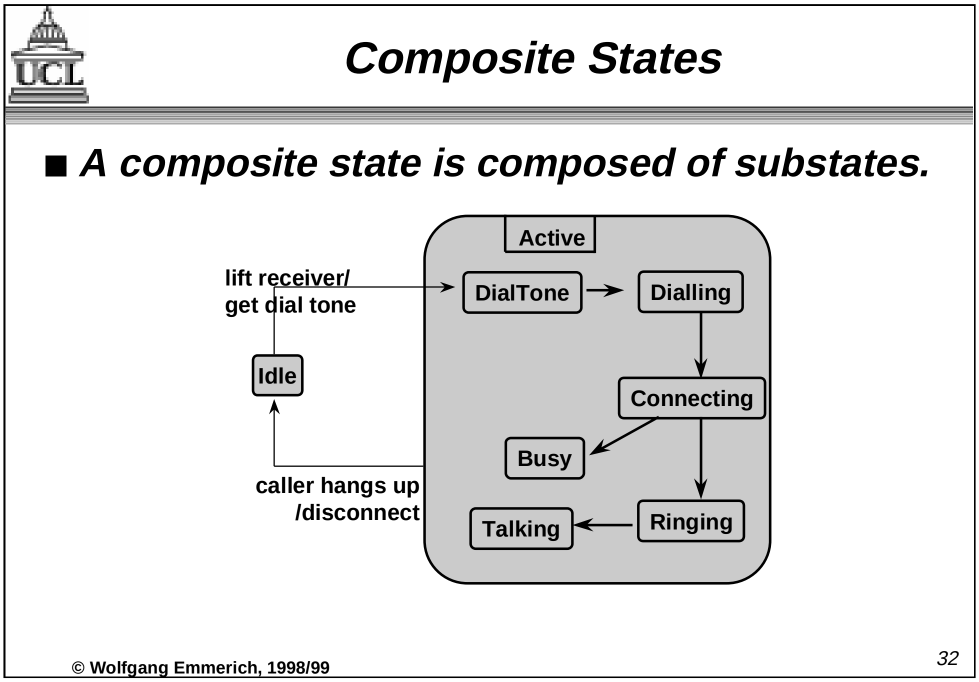
directory: home

entry/ display
message

© Wolfgang Emmerich, 1998/99

31

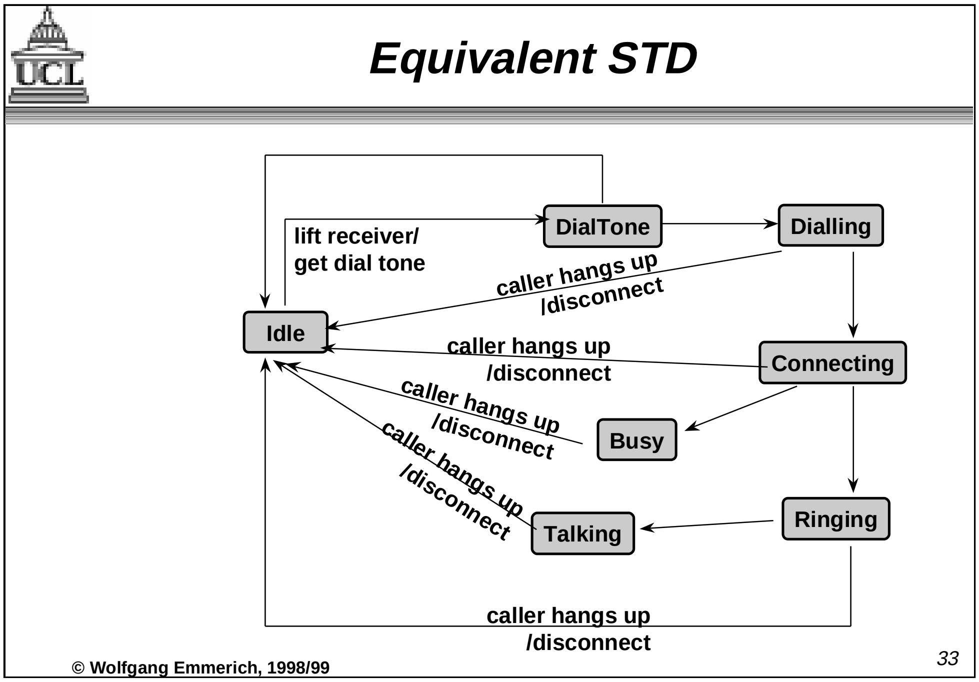
- Each state appears as a rounded box with the name of the state and optional state variables and triggered operations.
- A state variable is valid while the object is in the state and can be accessed and modified by operations within the state.
- Operations, called ‘actions’, must be non-interruptible. They can be implemented as private methods (a method is the implementation of an operation in an object-oriented programming language) on the controlling object. Operations can be preceded by the pseudo-event names, ‘entry’ and ‘exit’, effectively making the state into a self-contained module.
- The transition notation remains the same (directed arc), but the label has a more complicated syntax, of which the event name, with any associated parameters, is the principal component. Operations, possibly of other objects, can also be triggered by transitions and are included as the final component of the label. Other components are discussed below.
- States can also be composed of other states. For these composite states the composition is drawn within the node representing the state. An example of these composite states is shown on the next slide...



Composite States

- ***A composite state is composed of substates.***

- This slide displays the first example of the application of abstraction in state diagrams. At a high level of abstraction, there are only two states in the telephone, idle and active and two transitions between them. If a user lifts the receiver the telephone transits from idle to active and if the user replaces the receiver the system transits from active to idle.
- At a more concrete level of abstraction, however, an active telephone can be active in different ways. These are displayed in the refinement of state active. The receiver can play the dial tone and then the user can start dialing. After that, the telephone either plays the tone for busy or it tells the user that the phone of the desired partner is ringing and if the partner responds the connection will be established and the parties can talk to each other.
- The meaning of the composition of a state in this way is that the state 'active' is in exactly one of its substates.
- A substate inherits the properties of its composite state, variables and transitions. More precisely, outgoing transitions are inherited. This means that if the caller replaces the receiver the telephone will become 'idle', irregardless in which active state the telephone is.
- Note that state composition is the first way how Harel managed to reduce complexity. To clarify this and to explain the semantics of composition, the next slide displays a state transition diagram with the same semantics.



Equivalent STD

lift receiver/
get dial tone

Idle

DialTone

Dialling

Connecting

Busy

Ringing

Talking

caller hangs up
/disconnect

caller hangs up
/disconnect

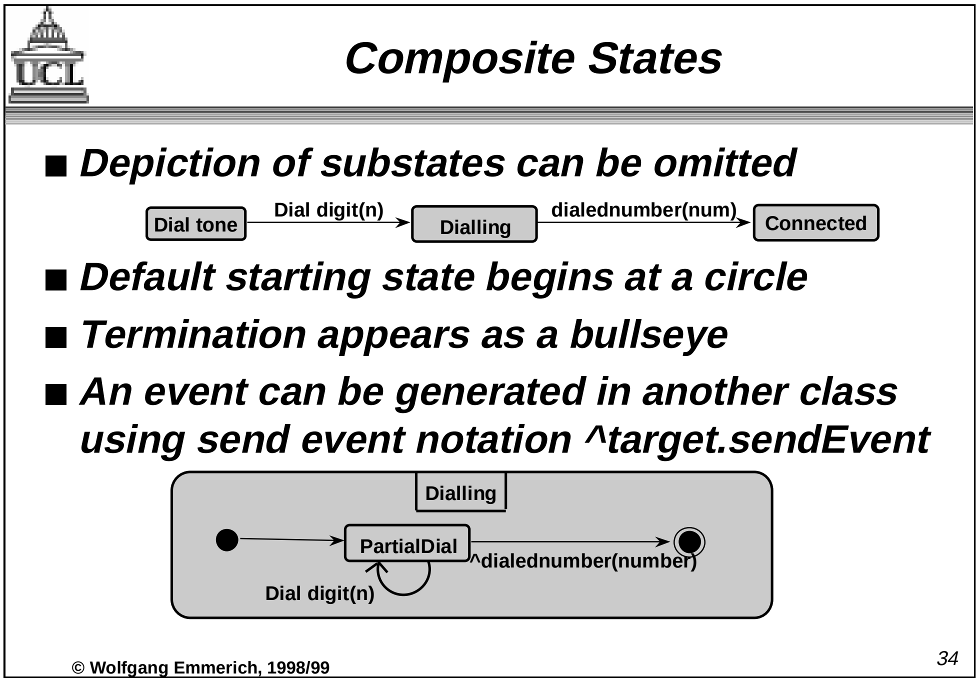
caller hangs up
/disconnect

caller hangs up
/disconnect

caller hangs up
/disconnect

33

- For reasons of simplicity, we have omitted the definitions of start and ending states in this diagram.
- Note that a number of additional transitions are necessary in the state transition diagram. These transition lead from each active state to the idle state. They were subsumed in the state chart under a single transition leading from the composite state 'active' to the state 'idle'.
- Hence composite states manage to reduce the number of transitions that are needed to model the behaviour of a class.
- The next slide displays that we are able to hide the complexity of a composite state completely...



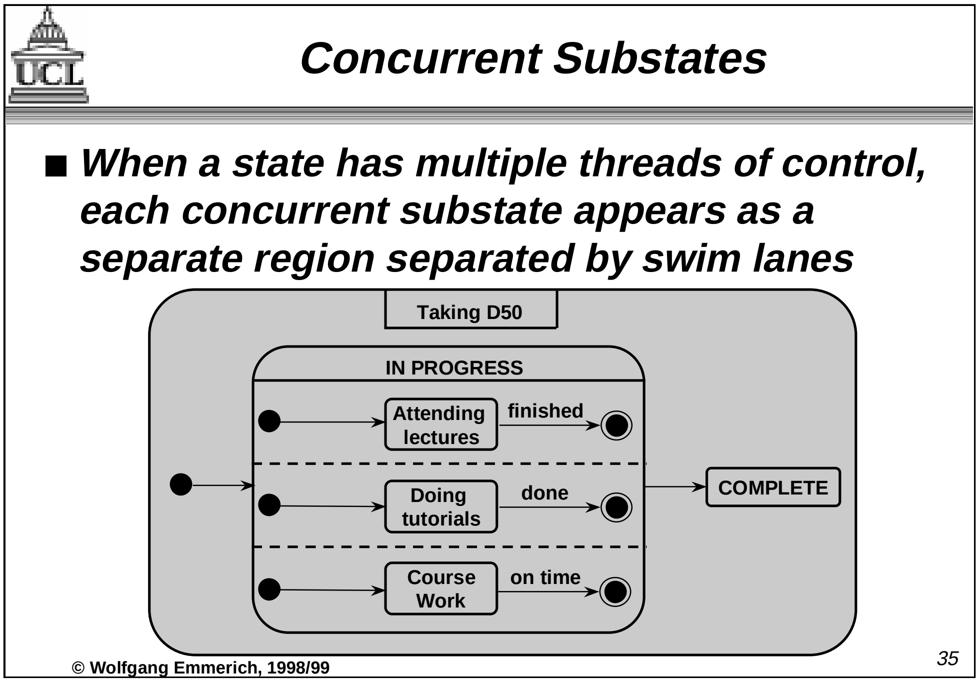
Composite States

- ***Depiction of substates can be omitted***

- ***Default starting state begins at a circle***
- ***Termination appears as a bullseye***
- ***An event can be generated in another class using send event notation ^target.sendEvent***

34

- Besides the notation we have used so far for composite states, there is another notation where we omit the substates of the composite state completely. The composite states are indicated just as if they were regular states and their definition is given in a separate diagram.
- This diagram needs to identify the substate that becomes active if the composite is activated. This is done by transition from a pseudo entry state that is represented as a filled circle and represents the activation of the composite state.
- We can use a transition to another pseudo state that represents the deactivation of the composite state and is represented by a bullseye. If we omit the bullseye, the state transitions defined for the composite state are inherited by all substates.
- On termination the composite state is shown sending an event to its higher-level self. In the ‘send event’ notation the ‘target’ is an expression designating a set of objects, which is not require in this example because it is fixed and well known.
- The composite states we have just introduced enable us to reduce the number of transitions needed in a state chart as we can define transitions between composite states that are then inherited by all its substates. There is, however, further potential for reducing complexity if we can manage to reduce the number of states needed. As the next slide shows, concurrent states achieve that...



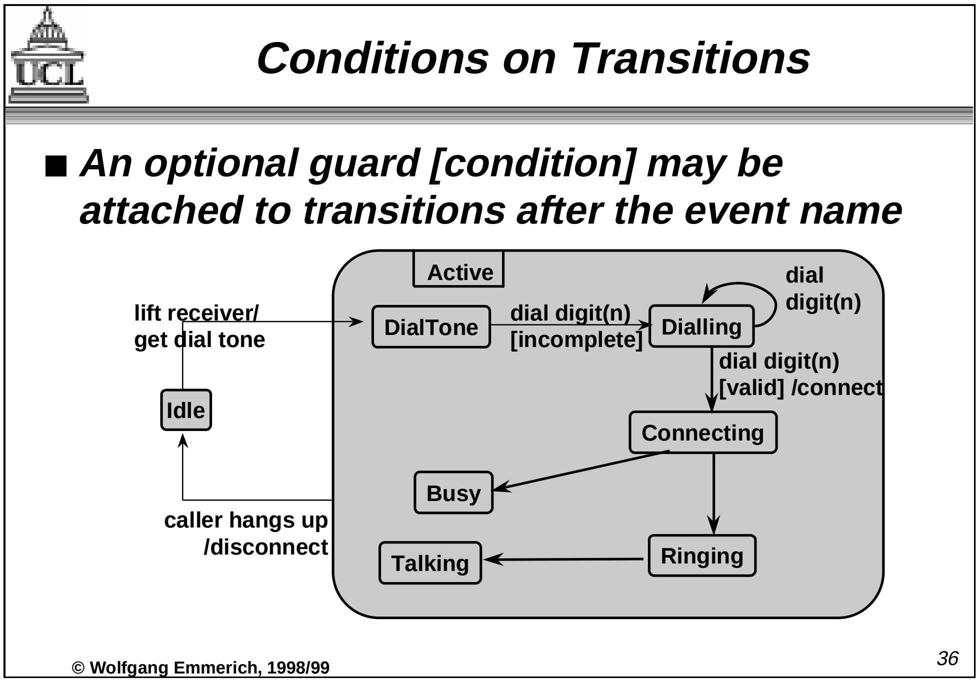
Concurrent Substates

- ***When a state has multiple threads of control, each concurrent substate appears as a separate region separated by swim lanes***

35

- The UML documentation suggests the following model of concurrency :
- An atomic object can be thought of as a finite state machine with a queue for incoming events. New events go on the queue until the object is free to deal with them. Composite concurrent objects contain several atomic objects as parts, each of which maintains its own queue and thread of control.

- A detailed view of a telephone diagram illustrates some other features of the UML notation



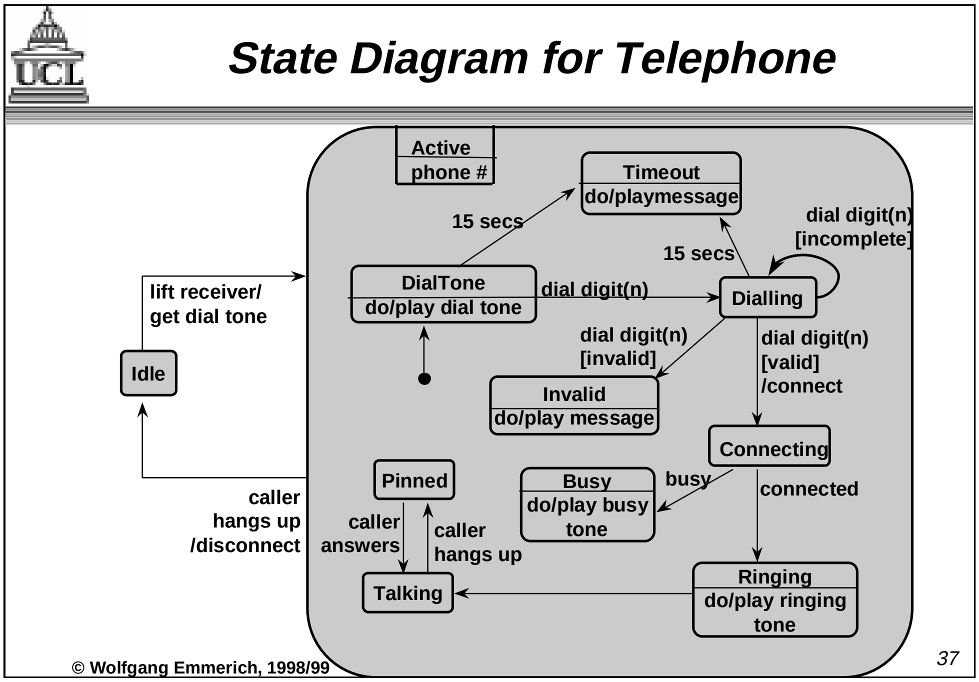
Conditions on Transitions

- *An optional guard [condition] may be attached to transitions after the event name*

© Wolfgang Emmerich, 1998/99

- A guard condition is a Boolean expression. If the event occurs and the expression is true, then the transition occurs, otherwise not. As in this example, two transitions can have the same name if different conditions are attached.

- The last of Harel's expansions of the original concept involved concurrency.



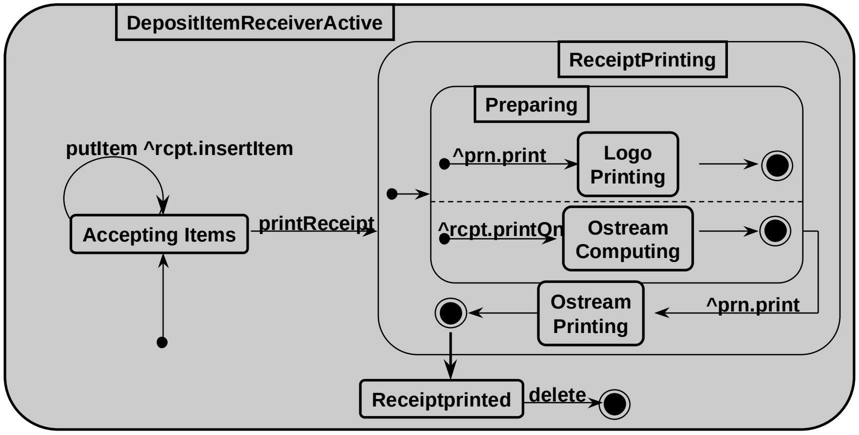
State Diagram for Telephone

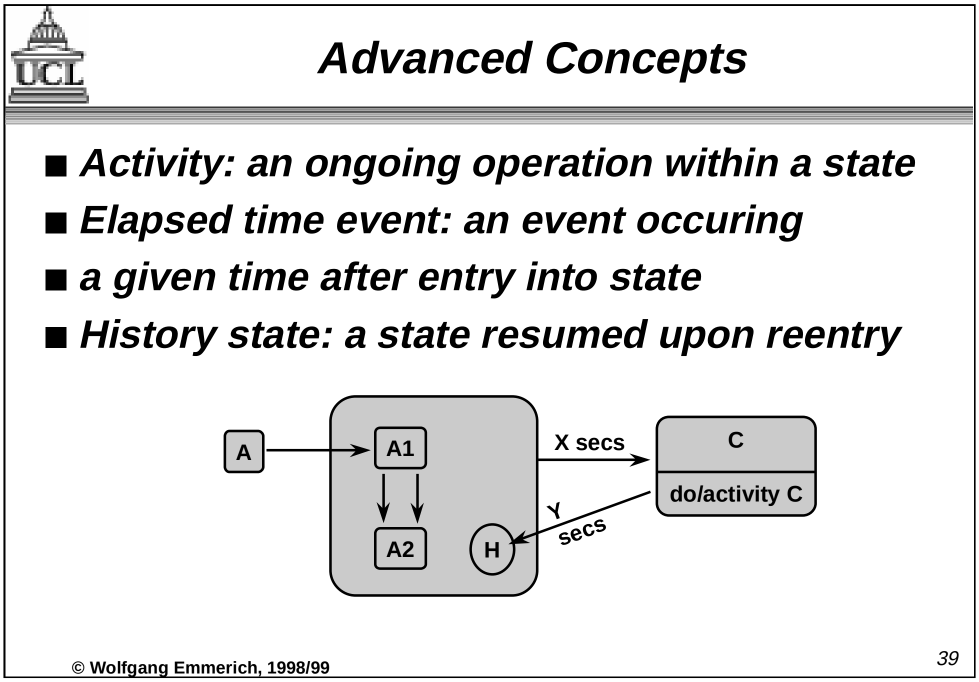
37

- ‘do/play dial tone’ denotes an ‘activity’.
- ‘15 secs’ is an ‘elapsed time event’.

Recycling Machine State Diagram

- ■ *One state diagram for each class*
- ■ *Example for Deposit Item Receiver:*



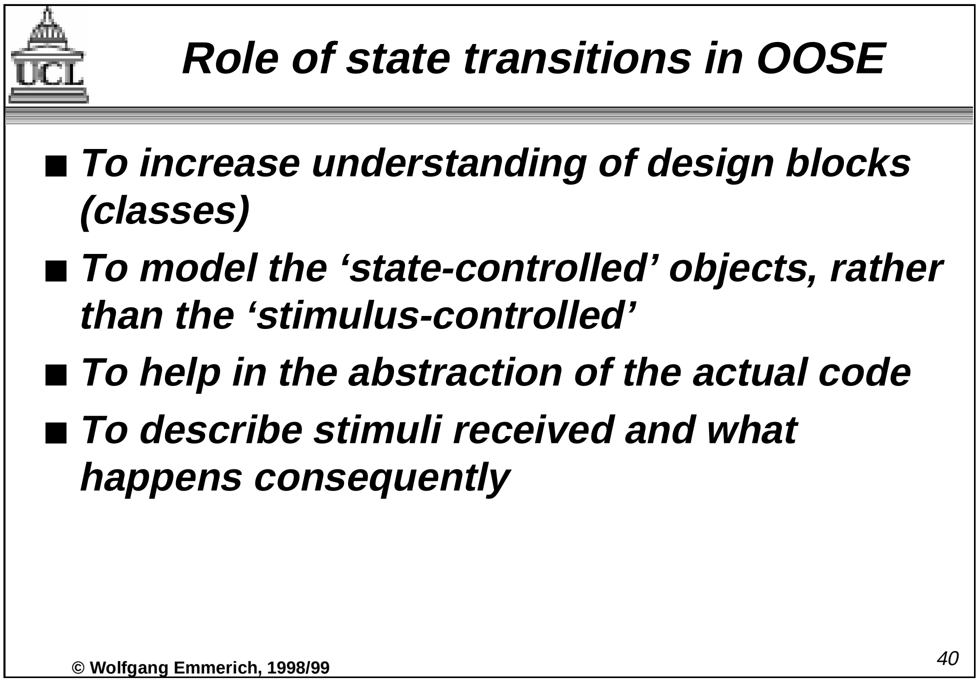


Advanced Concepts

- ■ *Activity: an ongoing operation within a state*
- ■ *Elapsed time event: an event occuring*
- ■ *a given time after entry into state*
- ■ *History state: a state resumed upon reentry*

A
A1
A2
H
X secs
Y secs
C
do/activity C

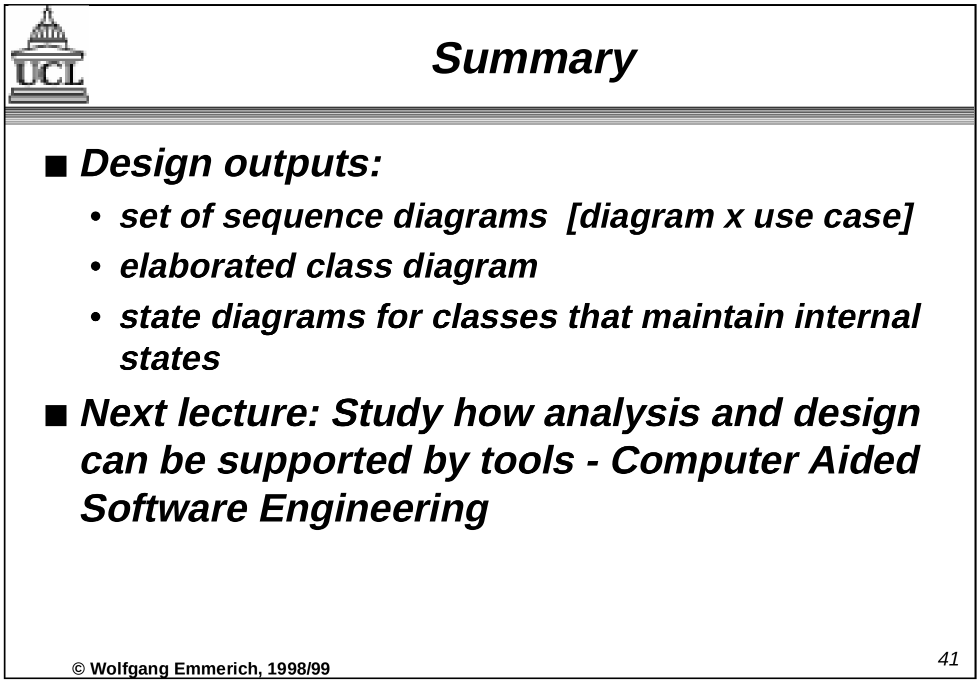
- An activity is an ongoing operation within a state that takes time to complete. It can be interrupted by an event that causes a state transition. An event causing exit forces its termination. An activity is indicated by a pseudo-event named ‘do’.
- In the case of the elapsed time event the sender is the “environment” rather than any individual object.
- The ‘history state’ (indicated by an ‘H’ within a circle) provides the means for a state to “remember” its substate when exited and to be able to resume the same substate on reentry into the state.



Role of state transitions in OOSE

- *To increase understanding of design blocks (classes)*
- *To model the ‘state-controlled’ objects, rather than the ‘stimulus-controlled’*
- *To help in the abstraction of the actual code*
- *To describe stimuli received and what happens consequently*

- During the block design stage of OOSE Jacobson recommends the examination of the states and state transitions of classes as a means of increasing understanding without going down to the actual code level.
- Changes of state are important in those objects whose response to stimuli depend not only on the stimuli but also upon their state on receipt. Such objects are called ‘state controlled’ and are more likely to have been modelled as the ‘control objects’. On the other hand the ‘stimulus-controlled objects’ will perform the same operation independent of state when a particular stimulus is received, e.g. the entity object ‘Deposit Item’.
- Jacobson does not consider the actual technique used as critical, so long as it meets the objective of helping the abstraction of code.
- This implementation objective affects the characteristics to be described, the stimuli received and the reactions that occur on receipt, which in the notation used in OOSE employ a variety of graphic symbols far richer than those in the UML equivalent.



Summary

- **_Design outputs:_**
  - ***set of sequence diagrams [diagram x use case]***
  - ***elaborated class diagram***
  - ***state diagrams for classes that maintain internal states***
- **_Next lecture: Study how analysis and design can be supported by tools - Computer Aided Software Engineering_**

- The set of sequence diagrams provides the means to elaborate the class diagram, particularly in terms of operations that are needed in class interfaces.
- In considering the next steps, we need to consider both the ‘system in use’ (as represented in use case and already defined in sequence diagrams) and the ‘objects in the system’ and how each will evolve in response to extermal stimuli. While the class diagram and the sequence diagrams provided an external perspective, we now need to focus on the internal aspects of a class and we need to specify the effect of stimuli on attribute values of the class.
- State diagrams (the subject of the next lecture) provide the essential means of describing the dynamic behaviour of a class, via the temporal evolution of an object in response to interactions with other objects inside or outside the system.
- For your background reading we would suggest:
- [JCJÖ92]
- [Hare87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8(3):231-274 1987.

- *DESIGN MODEL*

- *Inputs:*
  - requirements specifications relating to implementation environment
  - analysis model class diagram
  - use case descriptions

- 16) Identify characteristics of implementation environment including: programming language primitives (requiring notation)

- 17) Duplicate analysis model class diagram to create initial design model class diagram and revise by:
  - ‘normalisation’ of class structure to provide implementable interfaces and coupling
  - ‘encapsulation’ at architectural level to provide functionally cohesive packages
- 18) Formal design of the flow of control by description of all stimuli sent between objects in: a sequence diagram for each use case

- 19) Definition of interface of each class by extracting all operations for a class from each sequence diagram

- 20) Definition of state diagrams for each class

- 21) Complete design model class diagram

- *Notations introduced:*
- sequence diagram
- state diagram

- *Outputs:*
  - sequence diagrams [diagram x use case]
  - state transition diagram [diagram x class]
  - complete design model class diagram