Lecture 5

Object-Oriented Software Engineering: Design Model - 1

Dr Neil Maiden
Dr Stephen Morris
Dr Wolfgang Emmerich

School of Informatics
City University

OOAD Design 1 *5. 1*

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
- Introduction to Design 2

OOAD Design 1 *5. 2*

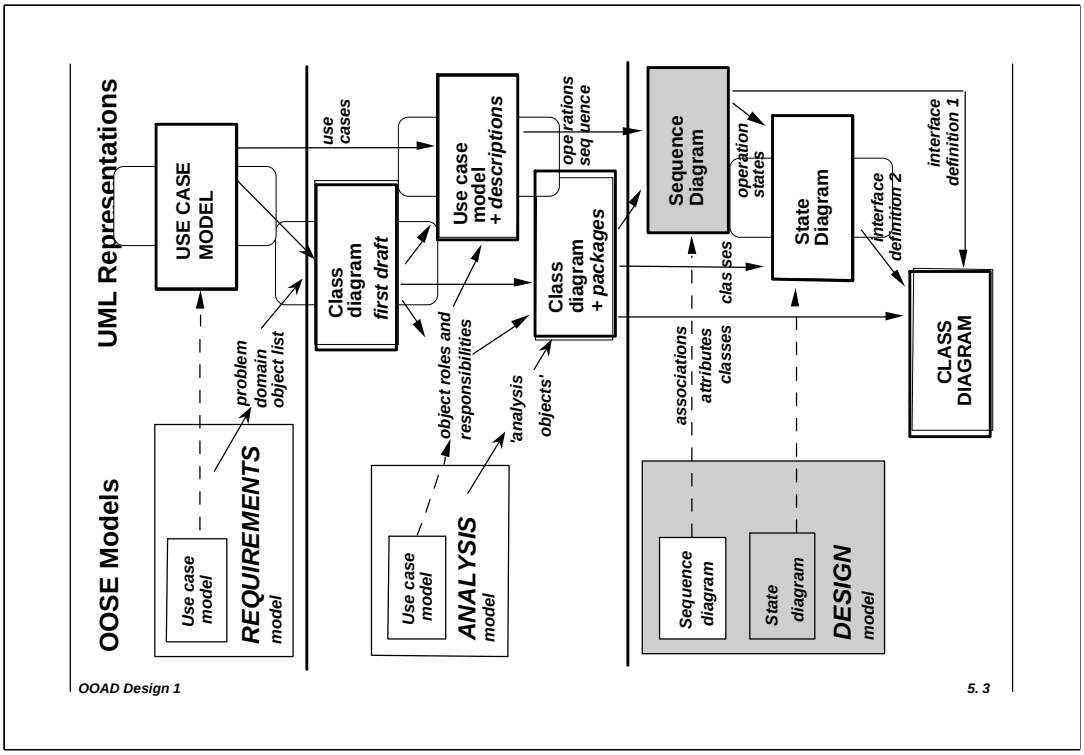
In this lecture, we are again following the model-based approach of Jacobson, though we aim at giving a more specific step-by-step approach, therefore we will be showing again the 'road map' relating the two views.

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.



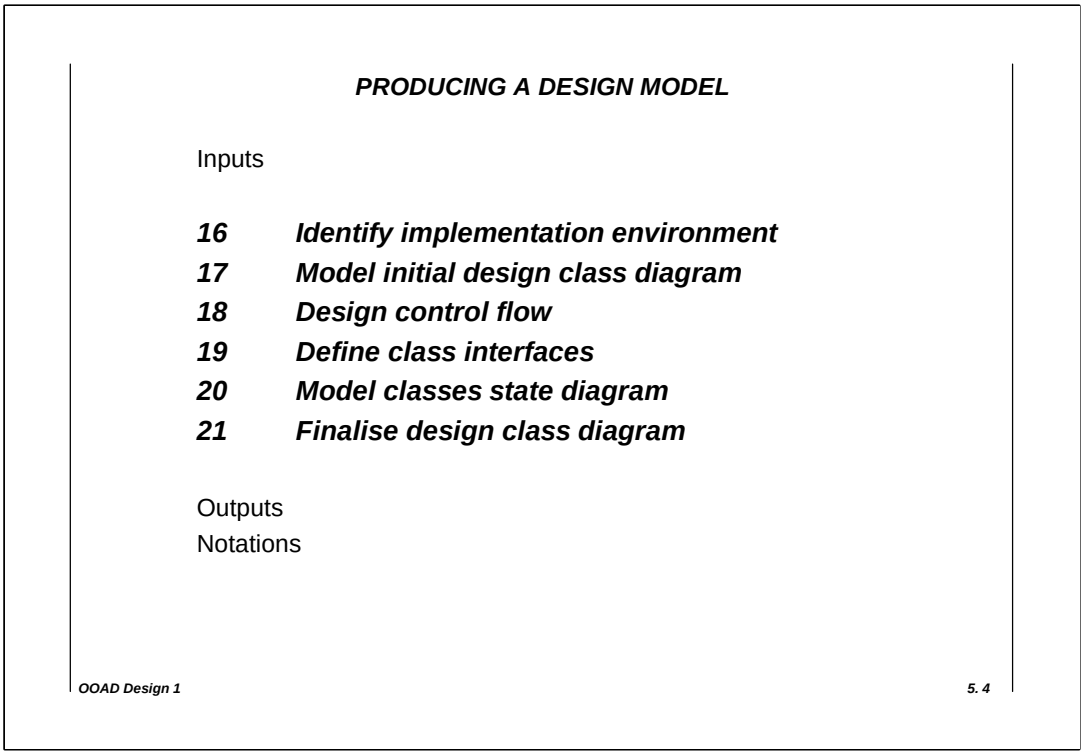
This slide shows our ‘road map’ again in order to highlight the aspects we discuss in this lecture and indicate how they fit into the overall object-oriented development process.

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



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.

[Complete version appears as notes to final slide]

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

OOAD Design 1 *5. 5*

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].

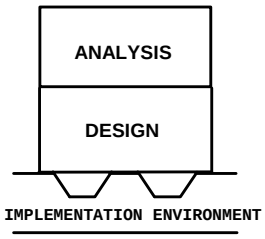
The most important output of the design stage will be a class diagram that is directly implementable in an object-oriented programming language. Moreover developers create a sequence diagram for each use case and a state diagram for each class.

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***



OOAD Design 1 *5. 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.

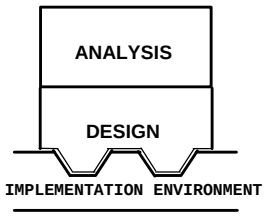
It might also be the case that existing class libraries (e.g. the Standard Template Library) can be reused and this will simplify the implementation. However, the design then has to identify classes in these libraries and the way in which they are to be deployed.

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 ***call through***

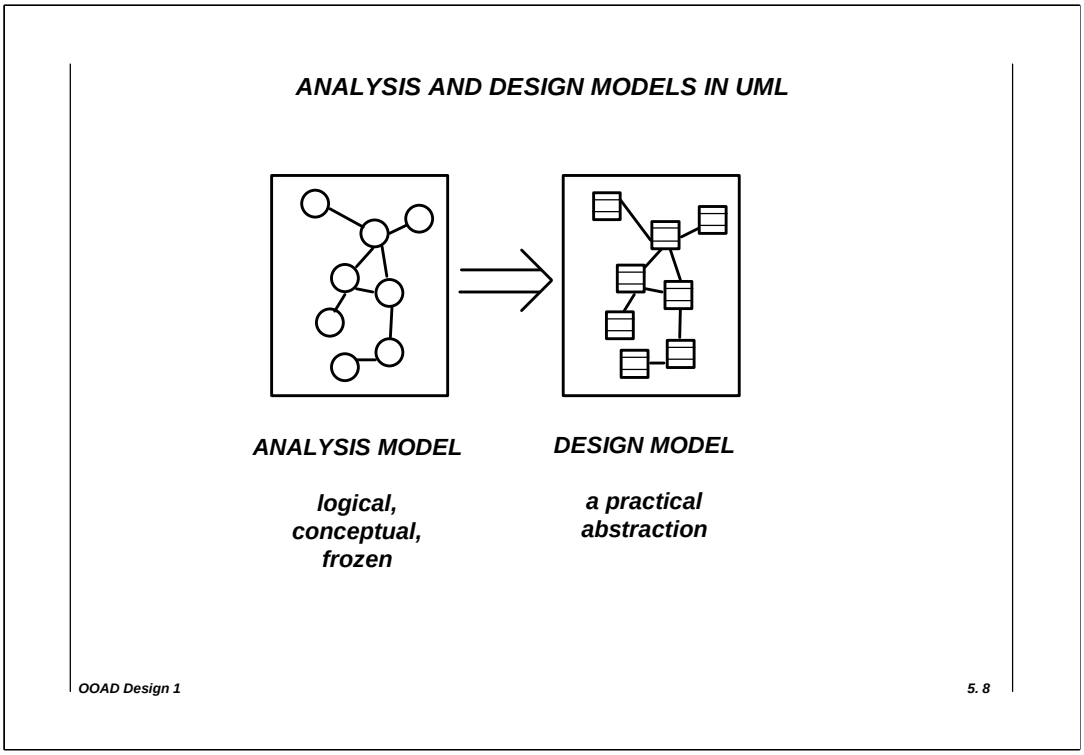


OOAD Design 1 *5. 7*

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...



- Semantic differences between models are:

  Analysis model

  • logical model

  • conceptual picture of system

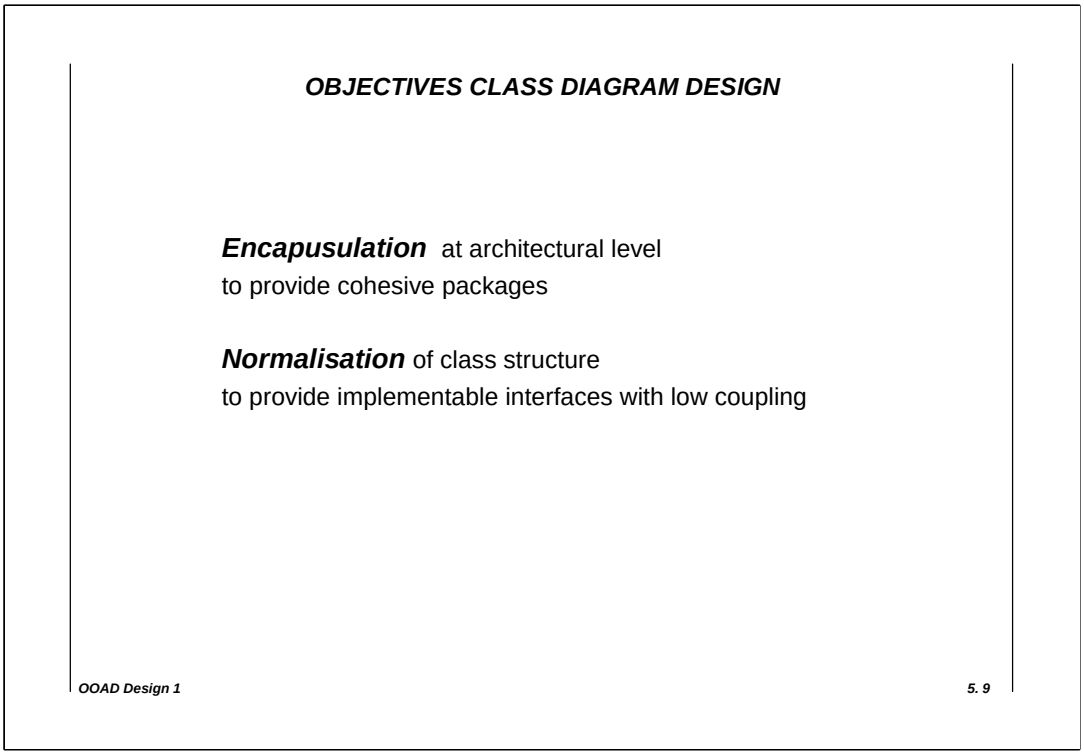
  • frozen at end of analysis process

  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.

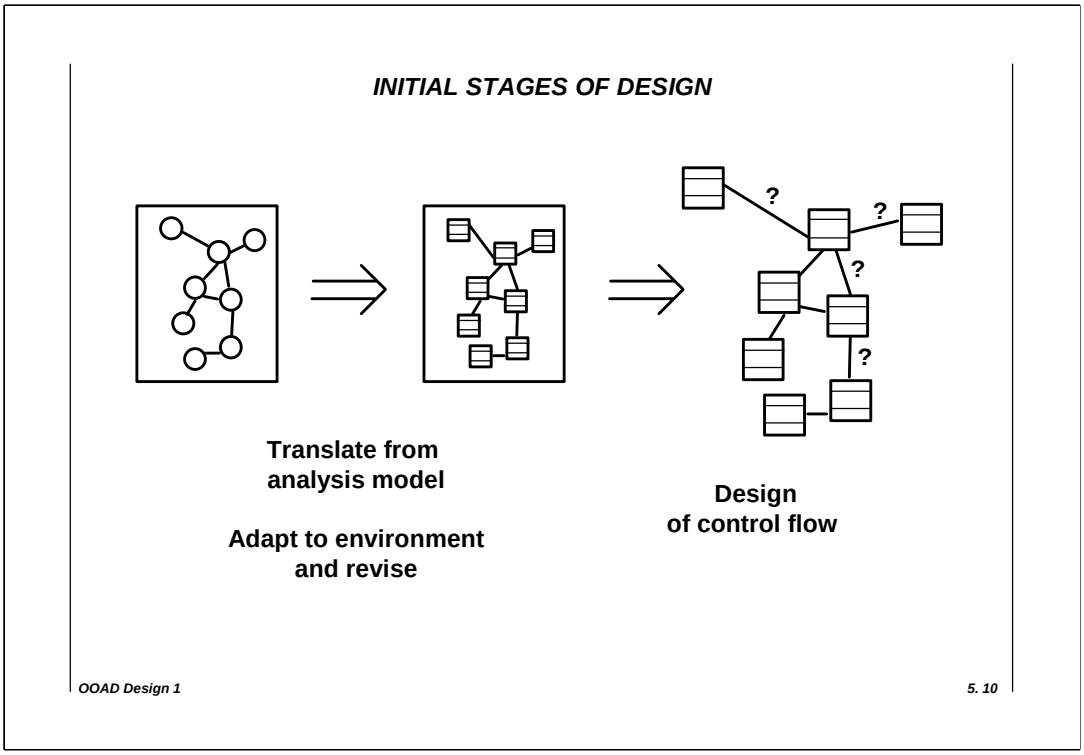


The transfer of the class diagram to a practical abstraction involves activities of 'encasulation', as defined earlier but at an architectural level. The aim of achieving encapsulation at an architectural level is to isolate dependencies on external components, such as a UIMS or a DBMS which are likely 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.

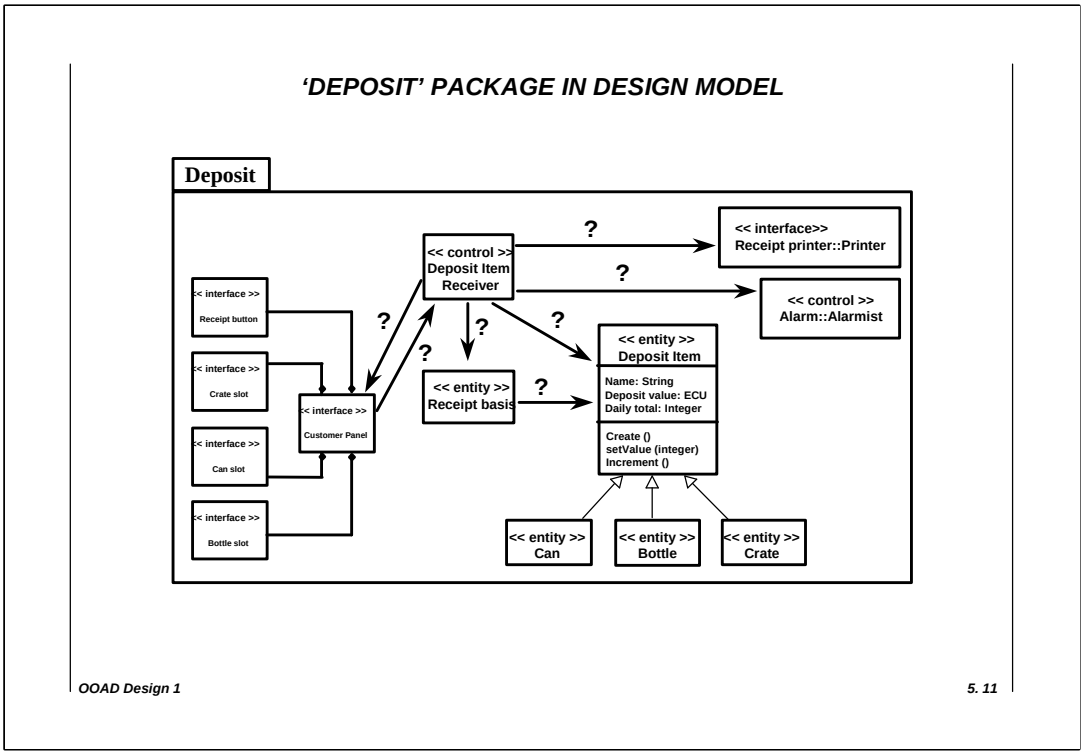


Having translated the analysis class model and adapted it to the environment and begun the necessary revisions, there follows a new form of design activity.

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.

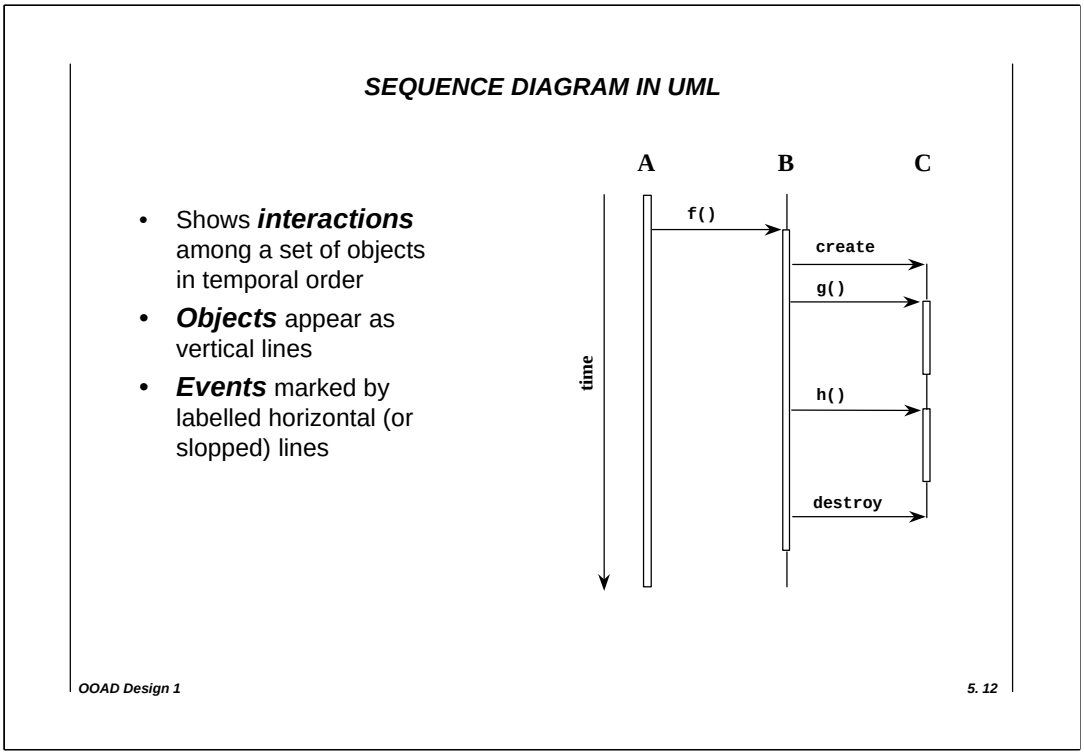
To achieve that we are going to model 'sequence diagrams' for each use case. Such a diagram identifies objects that occur within the use case and indicates the temporal order in which messages are exchanged between objects in order to stimulate operation executions.



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.



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.

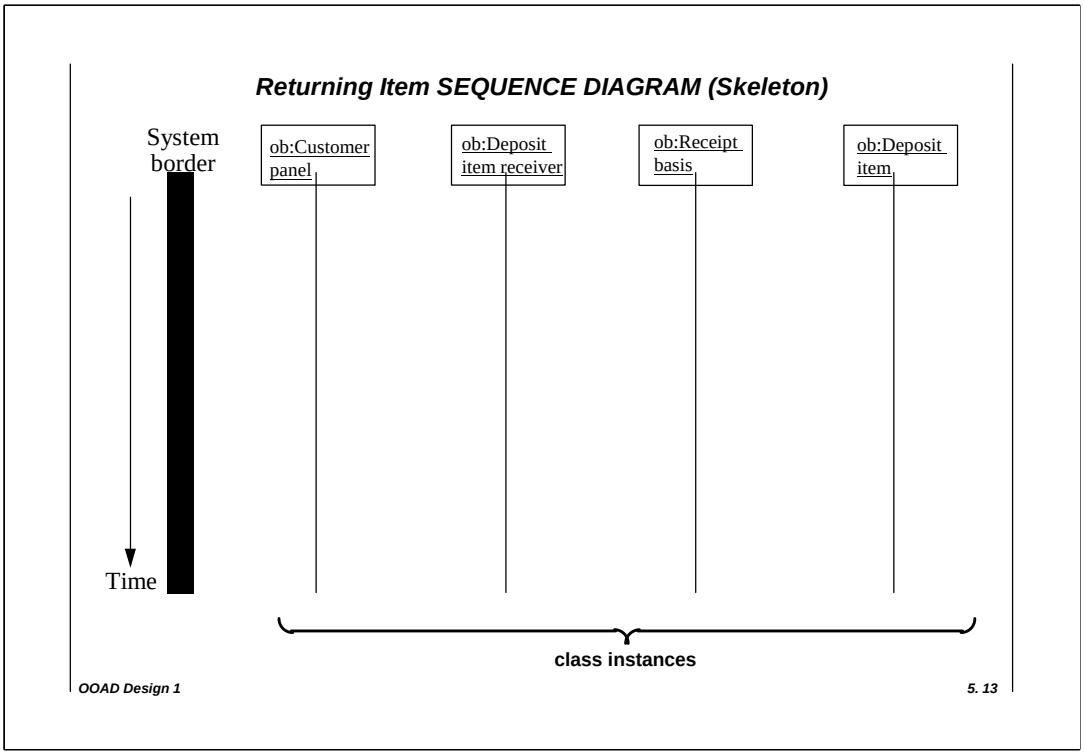
Objects appear as vertical lines. A very thin box (or a broader line) is shown when an object has the thread of control, otherwise the single line represents ‘created but in a waiting state’.

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).

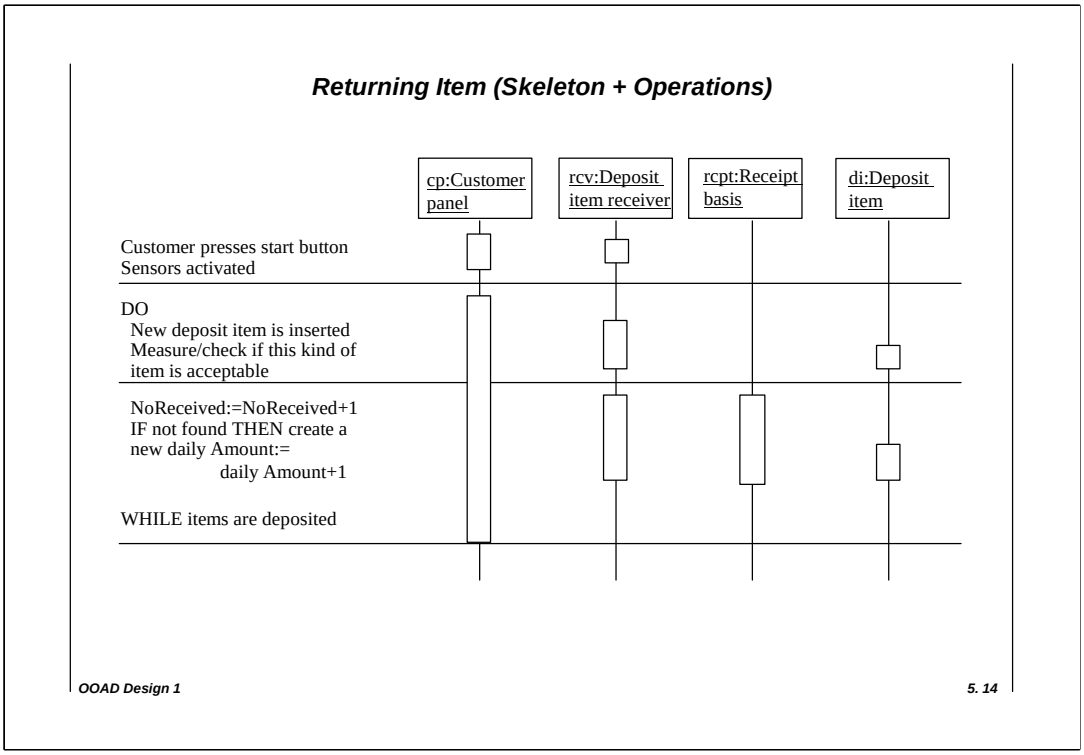
We now return to OOSE to see how sequence diagrams are used...



Sequence 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.

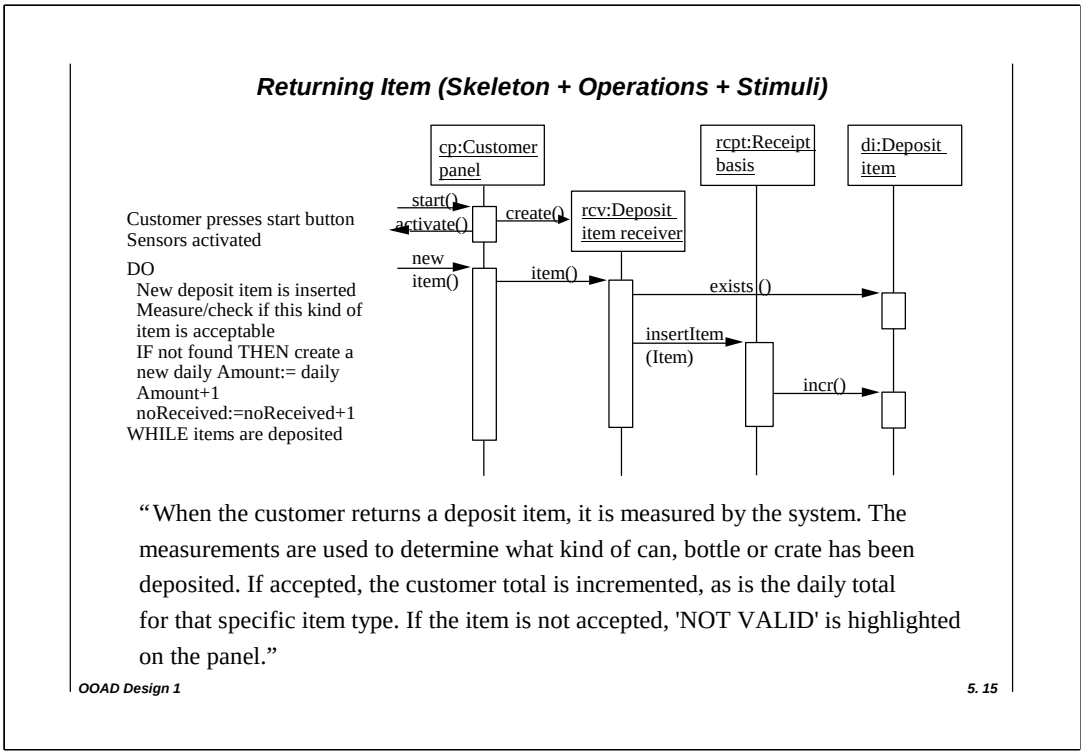
This slide displays merely the ‘skeleton’ of a scenario derived from the returning item use case that we have used as a running example. The sequence diagram will be continued on the next slide...



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...



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'. @@Where do rcpt and di come from??@@. 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

OOAD Design 1 *5. 16*

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.

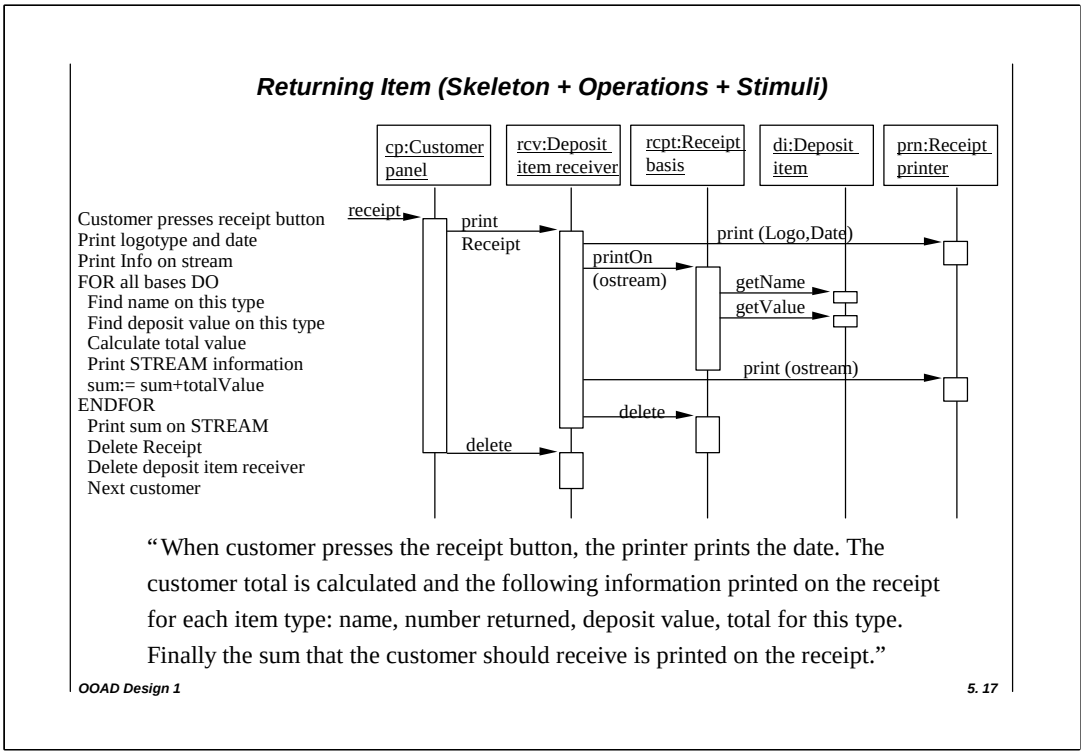
Also parameter lists should be kept as small as possible. Lengthy parameter lists are an indication that the operation is not really atomic but rather performs many different tasks. Then the operation should be split up into more simple operations. Simpler operations are easier to use and the probability 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.

Another ‘Returning Item’ scenario shows some of these points...

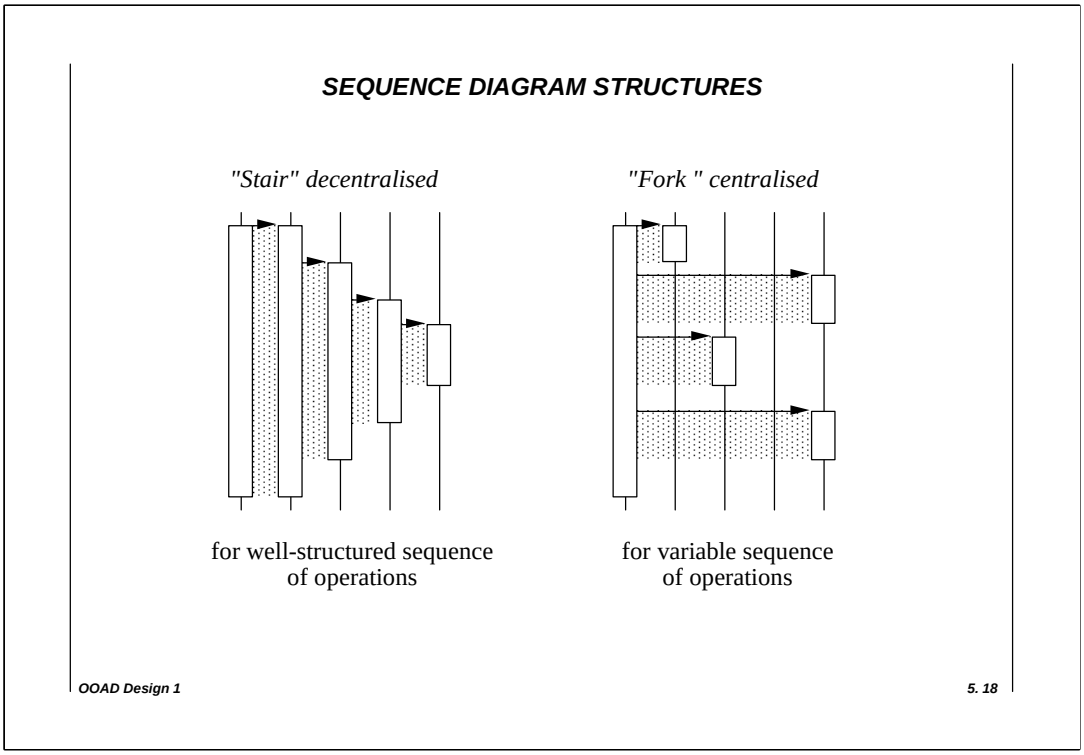


This sequence diagram originates in a returning item scenario where the customer prints a receipt button on the customer panel. Hence 'cp' will receive a 'receipt' event. The panel will delegate the printing of the receipt to 'rcv' by sending a 'printReceipt' event. The 'printReceipt' operation in 'rcv' will then stimulate the printing of the logo and the date on the receipt printer object 'prn'. It then stimulates execution of the 'printOn' operation of the receipt basis object 'rcpt'. The 'rcpt' object investigates the different types of deposit items that have been inserted and obtains the name and the value for each of the type from the deposit item object. Note that we have polymorphism here and that the concrete name and value is determined by subtypes of deposit item. The receipt basis object 'rcpt' will write a formatted receipt string onto the parameter 'ostream' that was provided by the item receiver. The receiver will then delegate the printing to the receipt printer object and pass the 'ostream' as an argument. After the receipt has been printed the temporal information managed for the particular customer can be deleted and that is why the 'rcv' and 'rcpt' objects are deleted.

As you can see from this example, this stage of the design is crucial because it provides the first opportunity to examine whether the design can be implemented, and hence is likely to raise many questions e.g. about specific messages creating and deleting objects.

The transformation of use case description to pseudo code description (shown in Jacobson's left hand column) is an important step but this will be a straightforward exercise for every programmer.

This example also illustrates one of two common structures that appear in this form of sequence diagram. These will be discussed on the next slide ...



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.

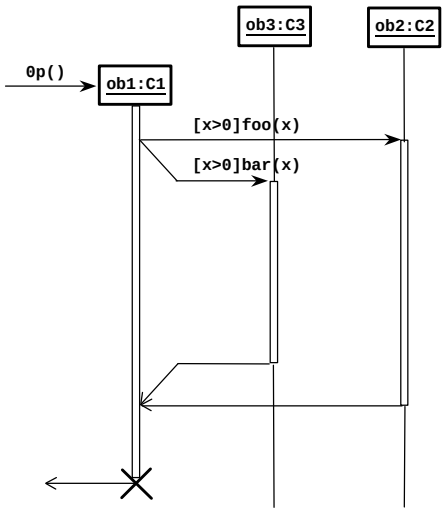
The right-hand side sequence diagram follows a centralised control pattern as the object displayed on the left of that diagram keeps full control and the objects that it stimulates return control immediately rather than stimulating other objects.

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 DIAGRAM + CONDITIONS

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



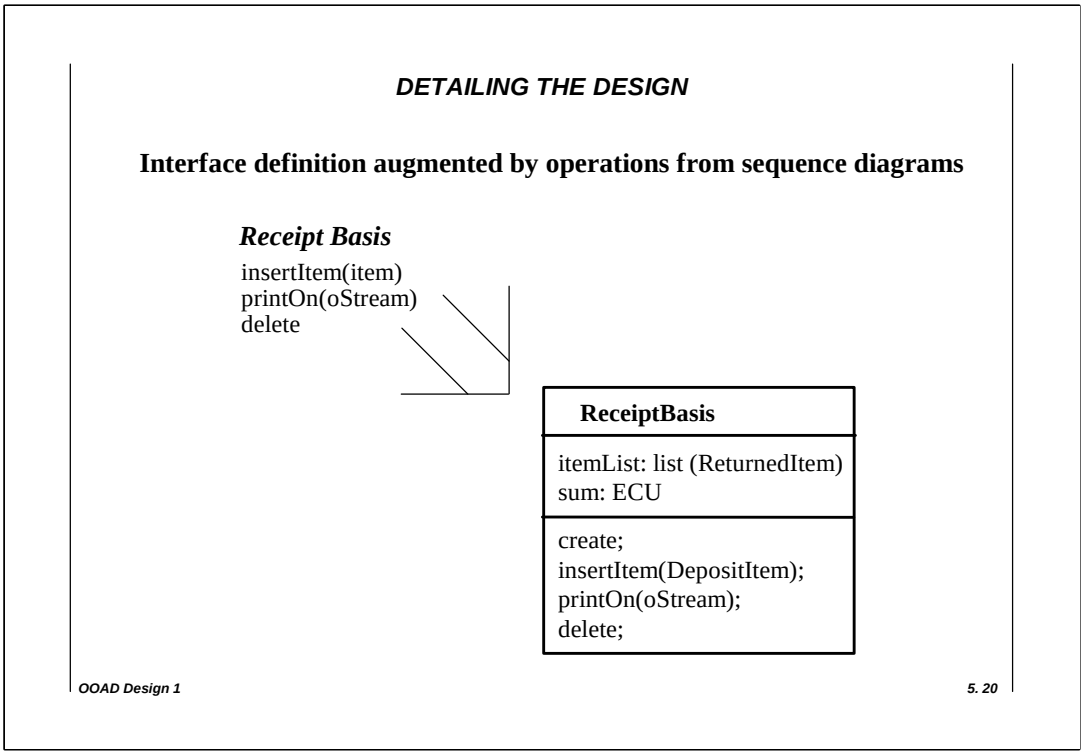
OOAD Design 1 *5. 19*

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).



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.

But 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.

SUMMARY

Design outputs so far :

- set of ***sequence diagrams*** [diagram x use case]
- elaborated ***class diagram***

Next steps :

- Need for further understanding of each class
- Development of ***state diagrams*** to define class behaviour

OOAD Design 1 *5. 21*

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.

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