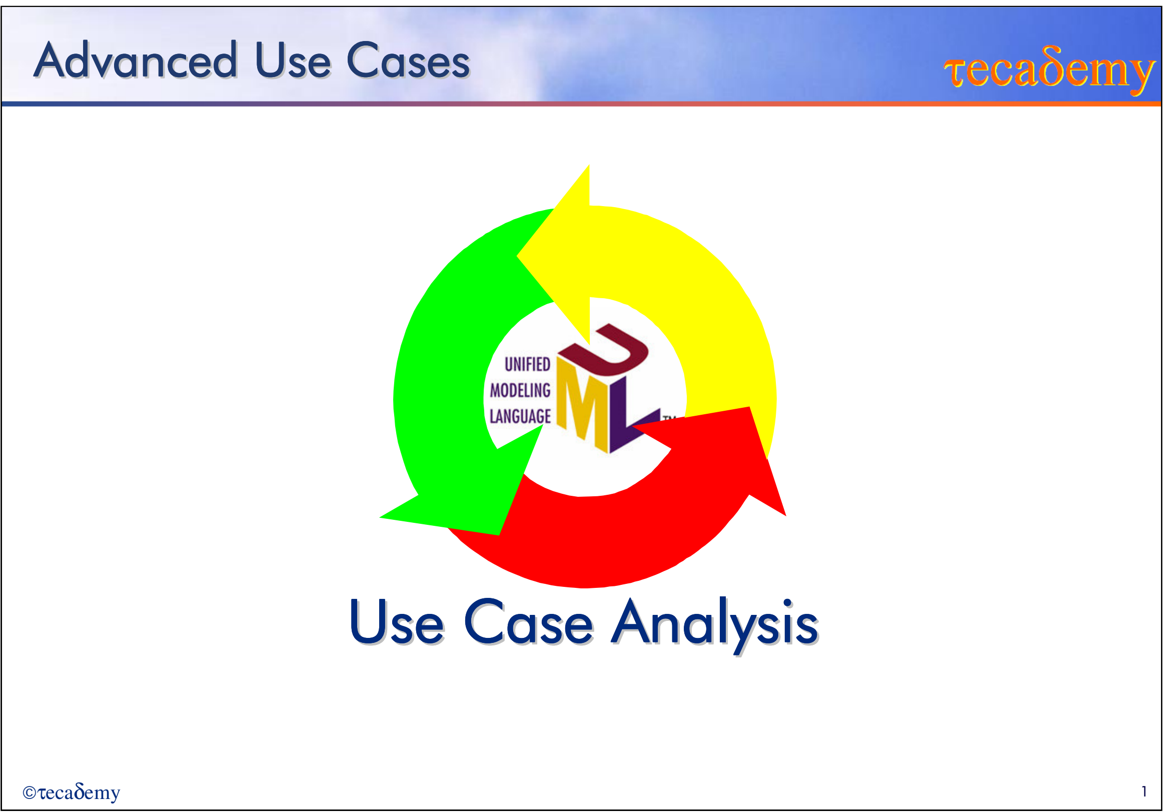
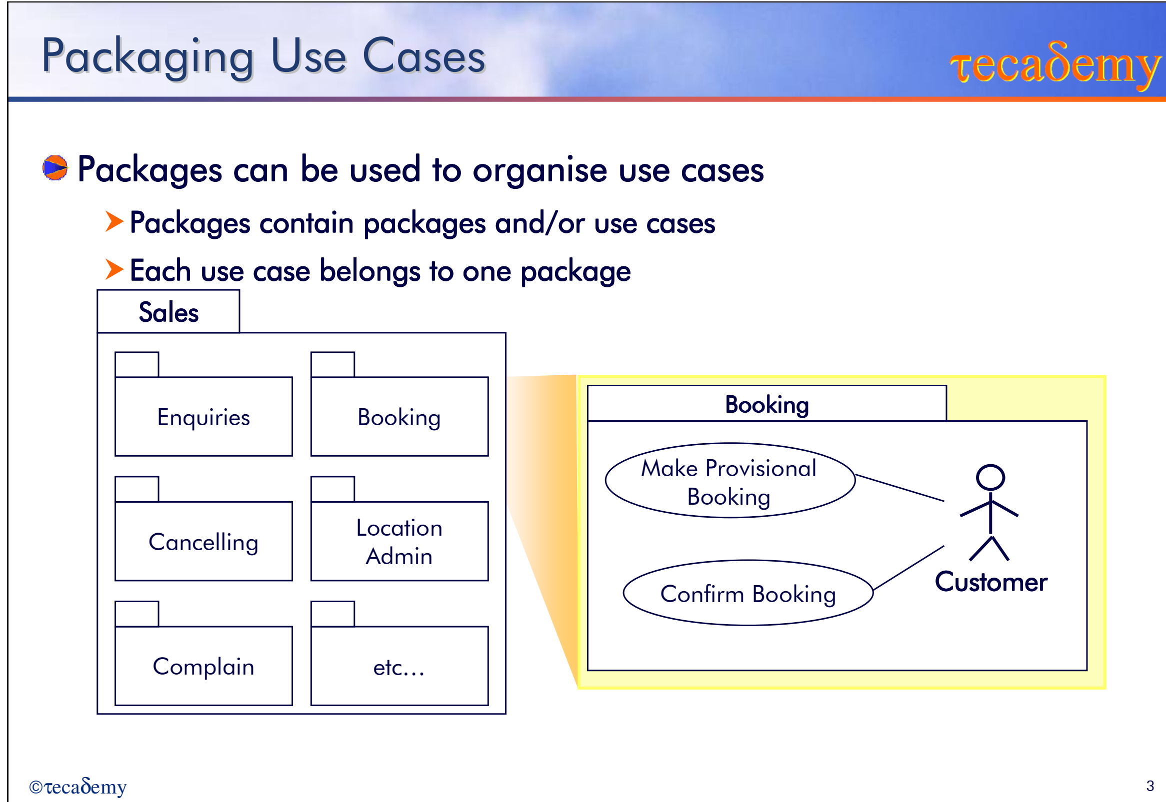

Advanced Use Cases
τecaδemy
UNIFIED
MODELING
LANGUAGE
UML
Use Case Analysis
©τecaδemy
1

Managing Complexity

τecaδemy

- Objectives
  - Familiarisation with techniques for further use case analysis
- Contents
  - Packaging
  - Generalisation and specialisation of actors
  - Inclusion
  - Extension
  - Generalisation and specialisation of use cases
  - Pre- and post-conditions
- Summary

©τecaδemy

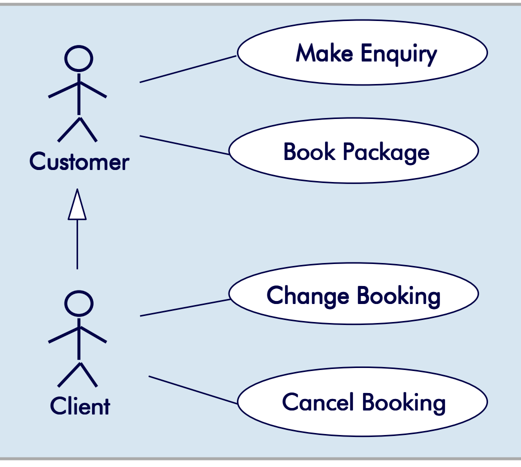


There may be a large number of use cases for a system, which may cause some problems when trying to determine whether the set of use cases is consistent and complete. The packaging mechanism of the UML provides a means of gathering together related use cases. Each lowest-level package can then be examined in relative isolation, to ensure that the use cases that it contains are consistent with each other, i.e. they neither overlap (which would allow functionality to be defined more than once), nor provide incomplete coverage (allowing functionality to be missed). Higher-level packages (such as **Sales** above), which contain mainly lower-level packages (e.g. **Booking**), are simply checked in the same way, but at a higher level of abstraction.

Actors Generalisation/Specialisation

tecaδemy

- Use generalisation/specialisation to classify actors
  - Generic actors have use cases common to all specialised actors
  - Specialised actors have special use cases



©tecaδemy 4

An actor may be specialised in order to allow consideration of a subset of the use cases of the generic actor. This simply divides up the use cases into smaller groups, so that each group can be considered independently. The generic actor is also useful for use cases that are common to all the specialisations, and again as a place holder in the use case context diagram, where the generic actor represents all the specialisations.

The separation of use cases between different specialised actors normally leads to some duplication in use cases, where the same interaction (or variations thereof) appears in similar use cases belonging to different specialised actors. This added complexity can be simplified by using the include, and extend relationships between use cases covered in the next few pages.

Use Case Inclusion

tecaδemy

- Extract a common sequence of events
  - From multiple use cases
  - Create a new inclusion use case
  - Modify the base use cases to include the inclusion

Book holiday

Base

Enquire re holiday

«include»

«include»

Check location list

Inclusion

1. The use case starts when the customer requests to book a course.
2. Include 'Check location list' use case.
3. If there are places available, . . .

1. For each course that the customer selects:
   a) the package run is identified to the system
   b) the system finds the package run location list
   c) the system checks for availibility

©tecaδemy 5

It is normal to find use cases which contain similar descriptions. To avoid redundancy and enhance reuse a common sequence of events can be abstracted out of a number of similar use cases and placed into a new use case referred to as an Inclusion use case. The original use cases can then have the common sequence of events replaced by an inclusion dependency which refers to the Inclusion use case. These original use cases are now referred to as Base use cases.

Experience shows that developers should wait to identify Inclusion use cases until a number of use cases with a common sequence of events have been described. Inclusion use cases should evolve from Base use cases, not the other way around.

Scenarios and sequence diagrams may be written for each use case, although within a Base use case the Inclusion use case is often summarised rather than appearing in detail.

An Inclusion use case may be included by many Base use cases. The Base use case may include many Inclusion use cases.

When graphically representing Includes use cases, a dashed arrow representing a dependency connects the Base use case to the Includes use case. The dependency has an «include» stereotype.

Use Case Extension

τecaδemy

- Base use case
  - Add extension points - locations where extensions are allowed
- Extension use cases
  - Specify the added behaviour at one or more extension points

Base — Book holiday

«extend»
[holiday full]

Extension — Offer alternate holiday

1. The use case starts when the customer requests to book a holiday.
2. Include 'Check location list' use case.
3. If a place is available create booking...
   extension point: [holiday full] 'Offer alternate holiday' is called...

extension 'Offer alternate holiday'
1. The location list for the next holiday is found...

©τecaδemy 6

Systems are complex and use cases are designed to highlight that complexity. However we must document that complexity in a simple understandable manner. The extension point construct in use cases allows us to highlight that some complexity may exist, but leave the discussion about that complexity to be provided elsewhere. This allows understanding of the use case at one level of abstraction, effectively ignoring complex detail. The understanding of the detail can then be layered on top of this basic understanding.

When writing a use case then, try to keep it as simple as possible, and wherever more detail is allowed but is not necessary, add an extension point. Extension use cases are really just specifications of detailed behaviour which can be inserted into the base use case at a specified extension point.

A base use case can have many extension points. An extension use case can extend one or more extension points. An extension point can have many extension use cases which extend it.

When graphically representing extension use cases, a dashed arrow representing a dependency connects the extension use case to the base use case. The dependency has an «extend»

stereotype, and is labelled with the extension point(s) which are extended. A guard condition in square brackets may be present, and indicates when the extension is activated.

Use Case Generalisation

tecaδemy

- Parent use case
  - May be concrete or abstract (abstract in italics)
- Child use case
  - A more specific form of the parent
  - Modifications of the parent behaviour are allowed throughout

Parent — *Pay for holiday*

Child — Pay for holiday By Credit Card

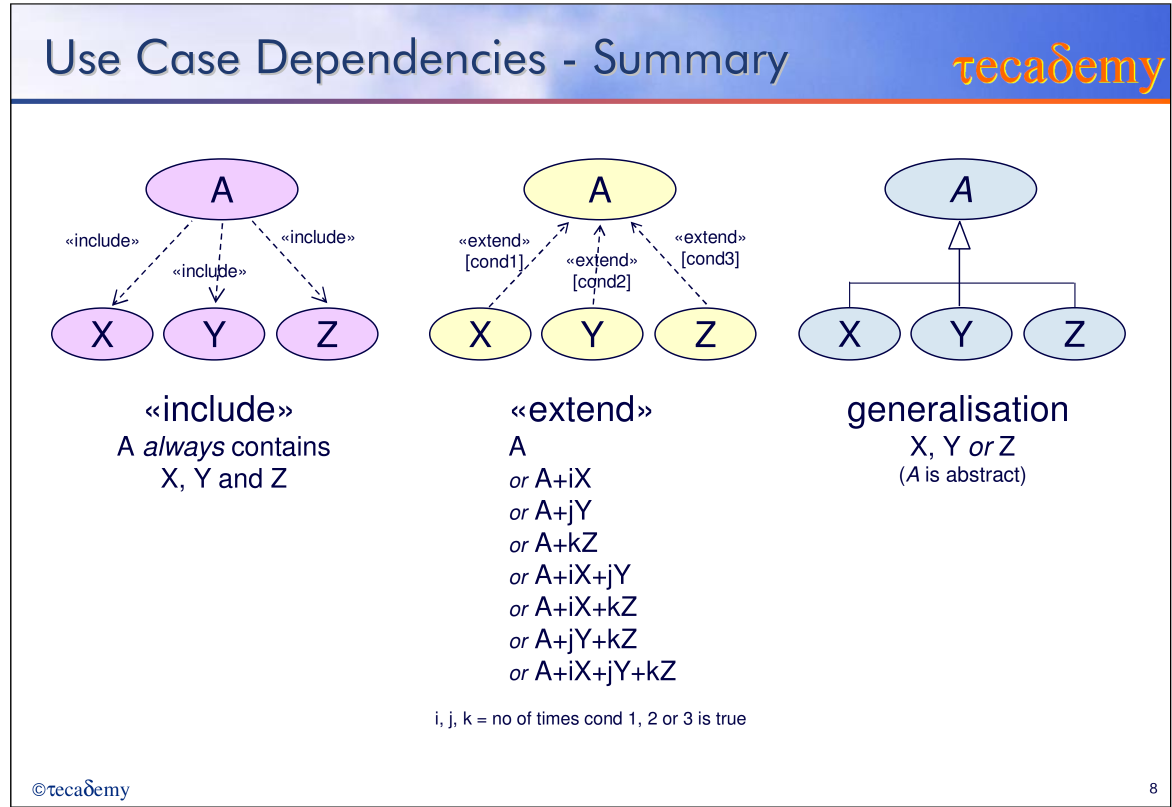
1. The customer chooses a payment method.
2. The details are checked, and recorded.

1. The customer chooses to pay by credit card.
2. The credit card details are read.
3. The system contacts the credit agency and requests payment of the required amount from the supplied card.

©tecaδemy 7

The generalisation relationship between use cases lets us capture the functional requirements of a complex system using layers of abstraction. We can describe a use case at an abstract or general level before supplying more concrete or special descriptions.

When graphically representing a generalisation relationship between use cases, a generalisation arrow (solid line, large empty arrowhead) points from the child use case to the parent use case.

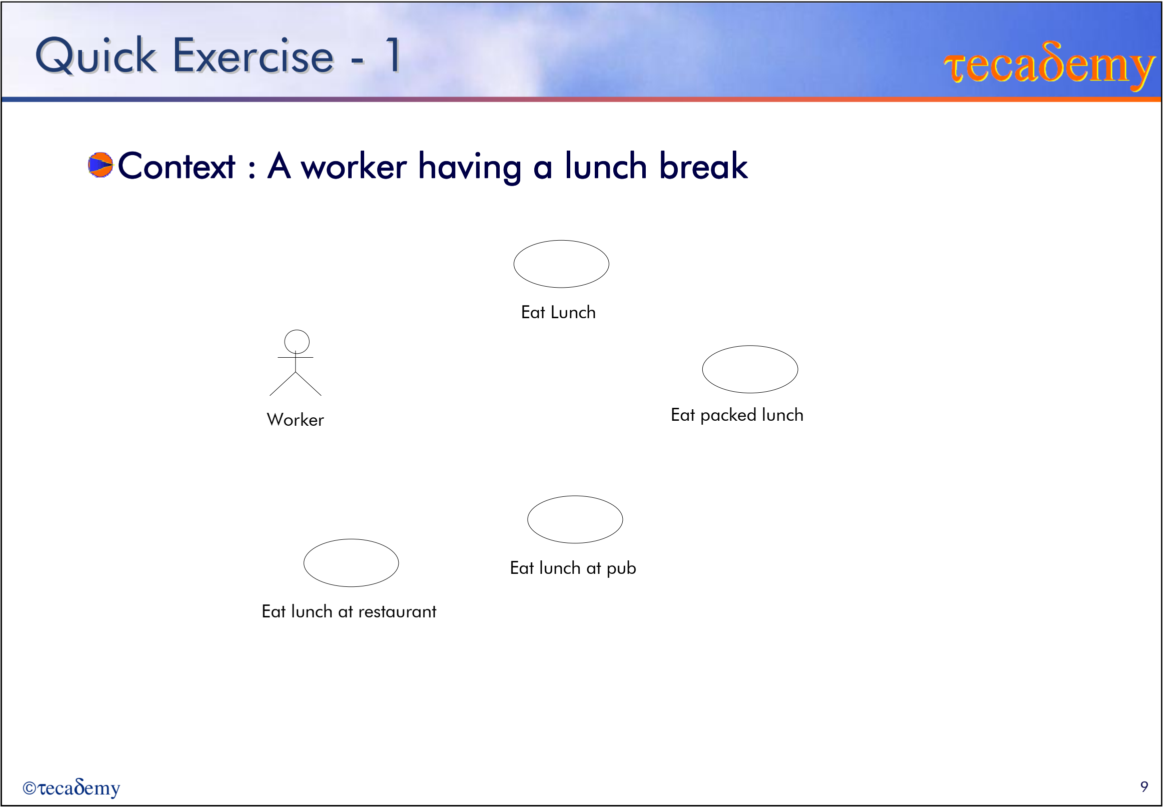


To summarise:

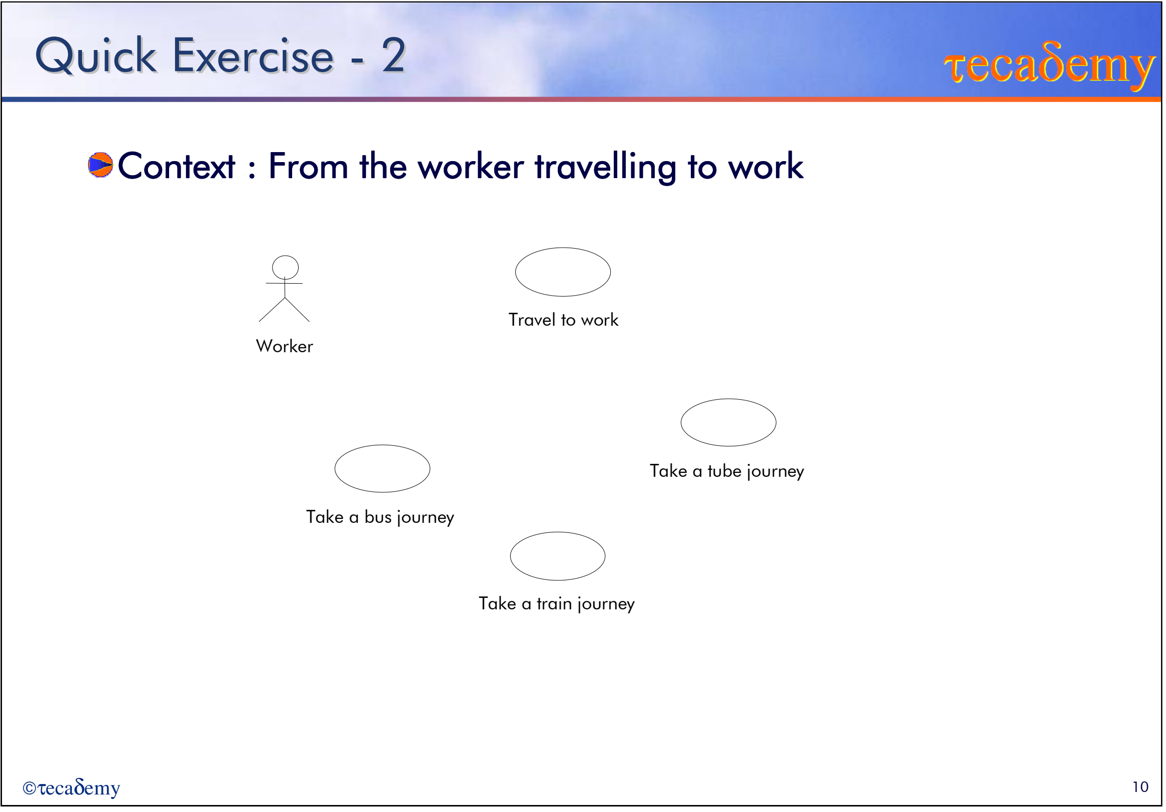
A «includes» X, Y, Z implies X, Y and Z are executed every time A is invoked. X, Y, Z can be included by other use cases. X, Y, Z can have their own inclusion or extension use cases. They can be specialised or generalised. They can have their own primary and/or secondary actors or they can be called exclusively by a base use case.

X,Y,Z «extends» A implies A will run alone unless one or more of the specified conditions is met e.g. if cond1 is true, the behaviour X will be inserted into A at the extension point specified in the use case description. «Extends» can be thought of as a conditional «includes». Commonly (but not exclusively) used to handle exception situations.

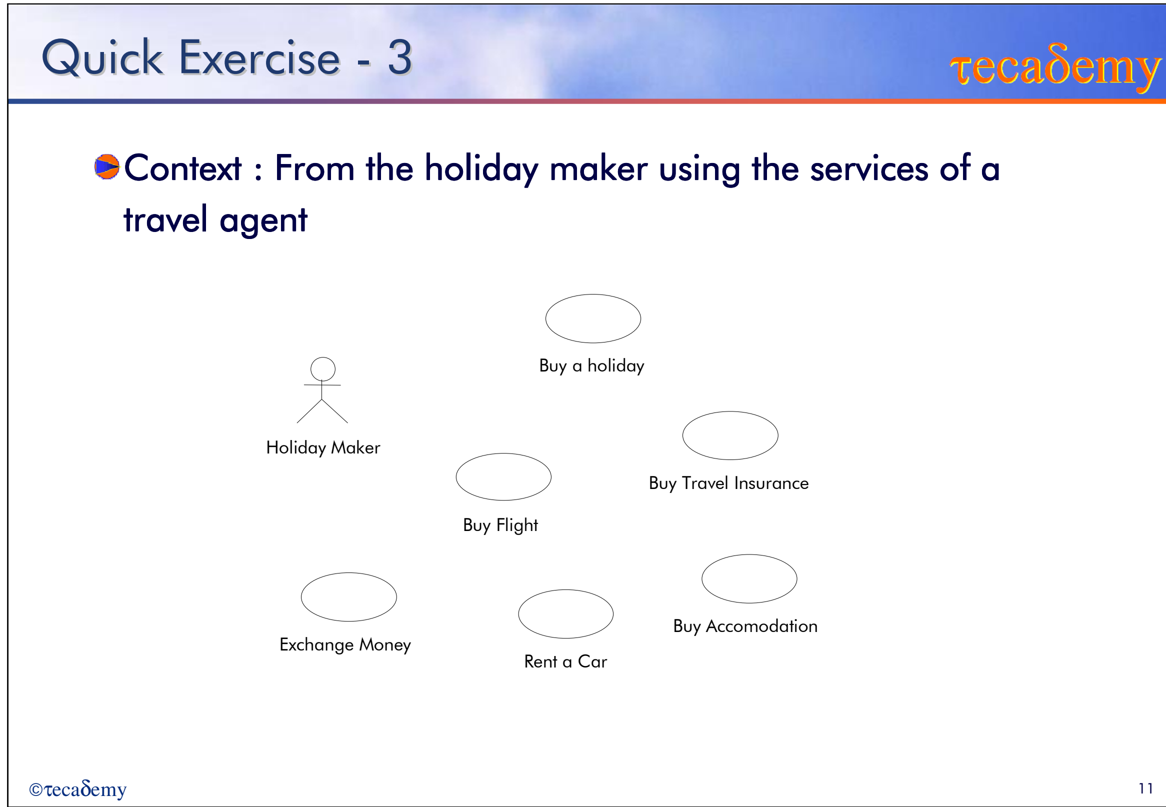
X, Y, Z are specialisations of *A* implies either X, Y or Z will be used whenever *A* is called.



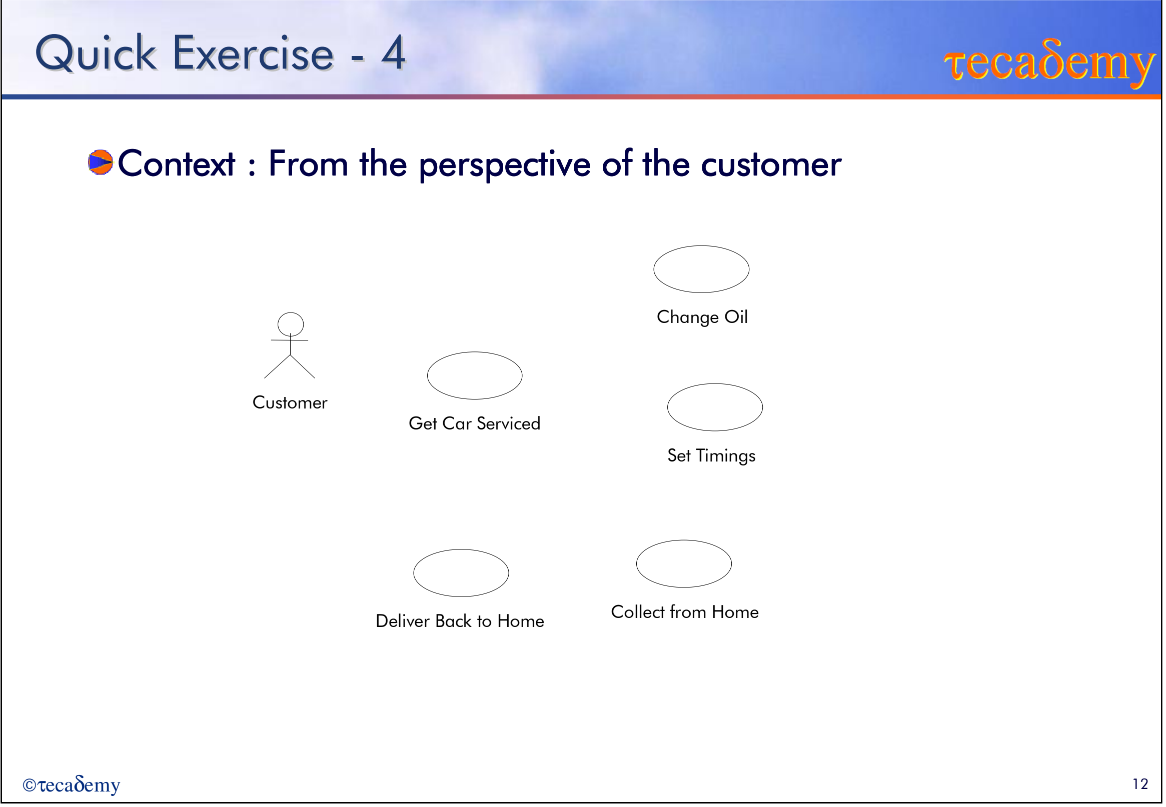
Assumptions: Workers only eat one lunch, They eat a packed lunch, pub lunch or at a restaurant.



Assumptions: All workers travel at least some part of the journey by foot. Workers may or may not travel for part of their journey by one or by a combination of bus, train or tube.



Assumptions: All holidays include flight and accommodation. Flights and accommodation cannot be purchased independently at this travel agent. Some holidays include car rental. This travel agent does not deal with car rental unless part of a package. Insurance can be bought with holidays or as a separate item. Money is always exchanged as a separate activity from buying a holiday.



Assumptions: Cars will always have oil changed and timings set. The customer can decide whether they wish to have the car collected and/or delivered.

Pre- and Postconditions

tecaδemy

- A precondition says "before this use case can be invoked, X *must* be true"
- A postcondition says "when this use case is finished, Y *must* be true"
- A pre- or postcondition represents a system *state*
- They can be used to check success or failure of a use case

©tecaδemy

13

When writing a use case description, it is important to identify the conditions under which the use case can be invoked (the *precondition*) and the condition the system is left in when the use case has ended (the *postcondition*). The pre- and post-conditions represent a state of the system – i.e. something that can be checked to be true. The difference between the two is representative of the change in the system as a result of the use case being run. They are best written as logical statements that can evaluate to a boolean 'true', although the degree of formality can vary from natural language (e.g. English) up to a formal specification languages (such as Z or VDM), depending on local conventions. The only rules are that they must be (a) unambiguous and (b) checkable within the context of the system. This is important as they will be used to determine whether or not the use case has been successful or not.

Pre- and Postconditions

τecaδemy

- A use case causes a *transition* from the precondition state to the postcondition state
- This can be shown on a *State Transition Diagram*

X is true
precondition
use case J
System states
postcondition
Y is true

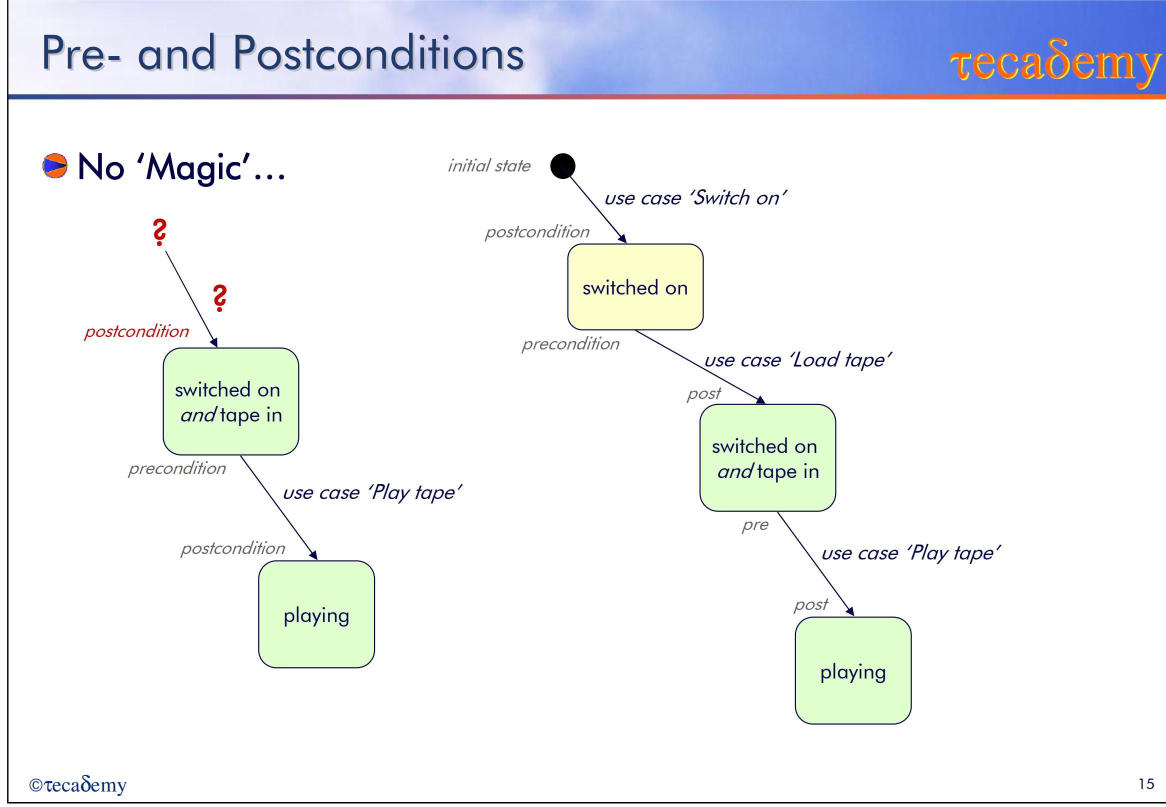
e.g.
switched on *and* tape in
precondition
use case 'Play tape'
VCR system states
postcondition
playing

©τecaδemy 14

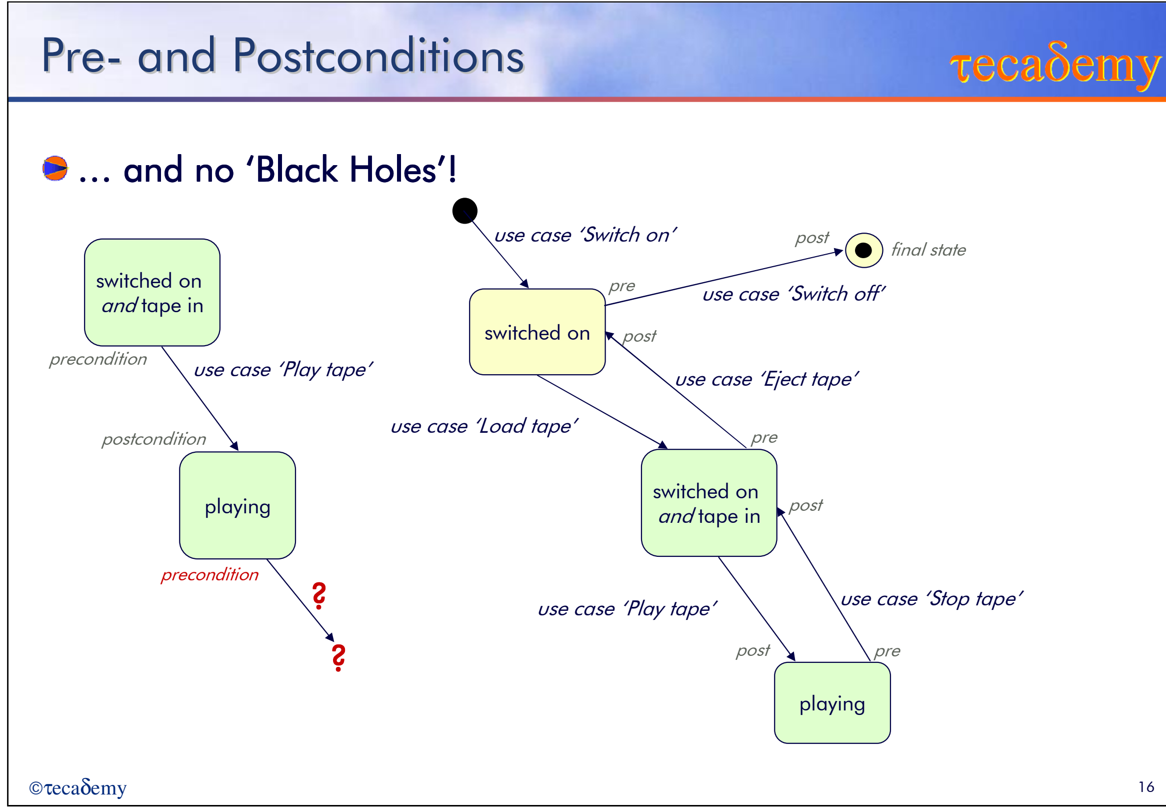
A use case, then, is an event which causes the system to move (or *transition)* from one state (precondition) to another (postcondition). For example, consider a VCR (video) system. An obvious use case would be 'Play tape' – but before we can do that, the VCR must be switched on *and* a tape must be loaded. If this isn't true, then pressing the 'play' button will have no effect. If it *is* true, however, then pressing the 'play' button will result in the motor starting, the tape head engaging and the VCR left in a 'playing' state.

This can be represented diagrammatically on a *State Transition Diagram* (STD), as seen above. States are represented as 'roundtangles', a transition from one state to another as an arrow, with the event or use case that causes the transition written alongside the transition arrow.

There are two rules with respect to STDs that can be very useful to Analysts concerned with ensuring no use cases are left undiscovered. Rule 1: *No Magic*. Rule 2: *No Black Holes*.



No Magic: one use case's precondition is another's postcondition. It is not possible to just 'be' in a state without having 'transitioned' there from another state. The only exception to this is the *Initial State* (represented by a black blob) which is the state the system is in prior to initialisation. For example, with reference to our VCR in its 'switched on and tape in' state, there must be at least one use case that gets us into that state – i.e. a use case which has 'switched on and tape in' *as a postcondition*, plus a precondition which we must now identify. Well, 'Load tape' is a reasonable candidate, and the VCR must be in a 'switched on' state before the tape transport mechanism can load the tape. This presents us with the same issue – how do we get to 'switched on'? We continue applying the 'no magic' rule until we identify the initial state, then we make sure we have descriptions written for each of the new use cases we might have discovered in the process.



No Black Holes: one use case's postcondition is another's precondition. It is not possible to just be *left* in a state without having an exit transitioned to another state. The only exception to this is the *Final State* (represented by a bull's eye) which is the state the system is in after shutdown. For example, with reference to our VCR in its 'playing' state, what is it that gets us out of 'playing'? Well, 'Stop tape' does, but that is not really a way out of the black hole, as it just gets us back to 'switched on and tape in' and a potential loop – now we need to identify which use cases will eventually take us to the final state. And then we need to write descriptions for them.

Pre- and Postconditions

tecaδemy

- STDs can be represented in table form for greater rigour

Use Case / State	Switch on/off	Load tape	Play tape	Stop tape	Eject tape	...
1. initial	2. Switched on					
2. Switched on		3. Switched on & tape in				
3. Switched on *and* tape in			4. Playing		2. Switched on	
4. Playing				3. Switched on & tape in		
5. final						

©tecaδemy

17

These diagrams can get very complex. It is arguable whether the benefits gained outweigh the effort expended, but when modelling safety- or mission-critical systems the arguments begin to move towards the need for greater, rather than less rigour. Even then, STDs in their graphical form do not guarantee that every possible combination of use cases and system states has been considered, a deficiency addressed by an alternative representation in tabular form, as shown above.

Across the table are written all the use cases under consideration; down the table are all the states identified from the pre- and postconditions of the use cases. It can be useful to number states, as this aids discussion. The table is filled in from the diagram – if the system is in the 'initial' state, and the 'Switch on' use case is run, the system is left in the 'switched on' state. And so on.

The empty cells, and there will probably be a few, reveal situations not covered in our analysis to date. Some will represent errors, or exceptions, or alternate successful combinations – some might be impossible.

Pre- and Postconditions

τecaδemy

- Detail can be added for all possible combinations…

Use Case / State	Switch on/off	Load tape	Play tape	Stop tape	Eject tape	…
1. initial	2. Switched on / display 'no tape'	1. initial	1. initial	1. initial	1. initial	…
2. Switched on	5. final	3. Switched on & tape in	2. Switched on / display 'no tape'	2. Switched on / display 'no tape'	2. Switched on / display 'no tape'	…
3. Switched on *and* tape in	5. final	3. Switched on & tape in / display 'error'	4. Playing	3. Switched on & tape in / displ 'stopped'	2. Switched on	…
4. Playing	do/ Stop tape 5. final	4. Playing / display 'error'	4. Playing	3. Switched on & tape in	do/ Stop tape 3. Switched on & tape in	…
5. final	2. Switched on	5. final	5. final	5. final	5. final	

©τecaδemy 18

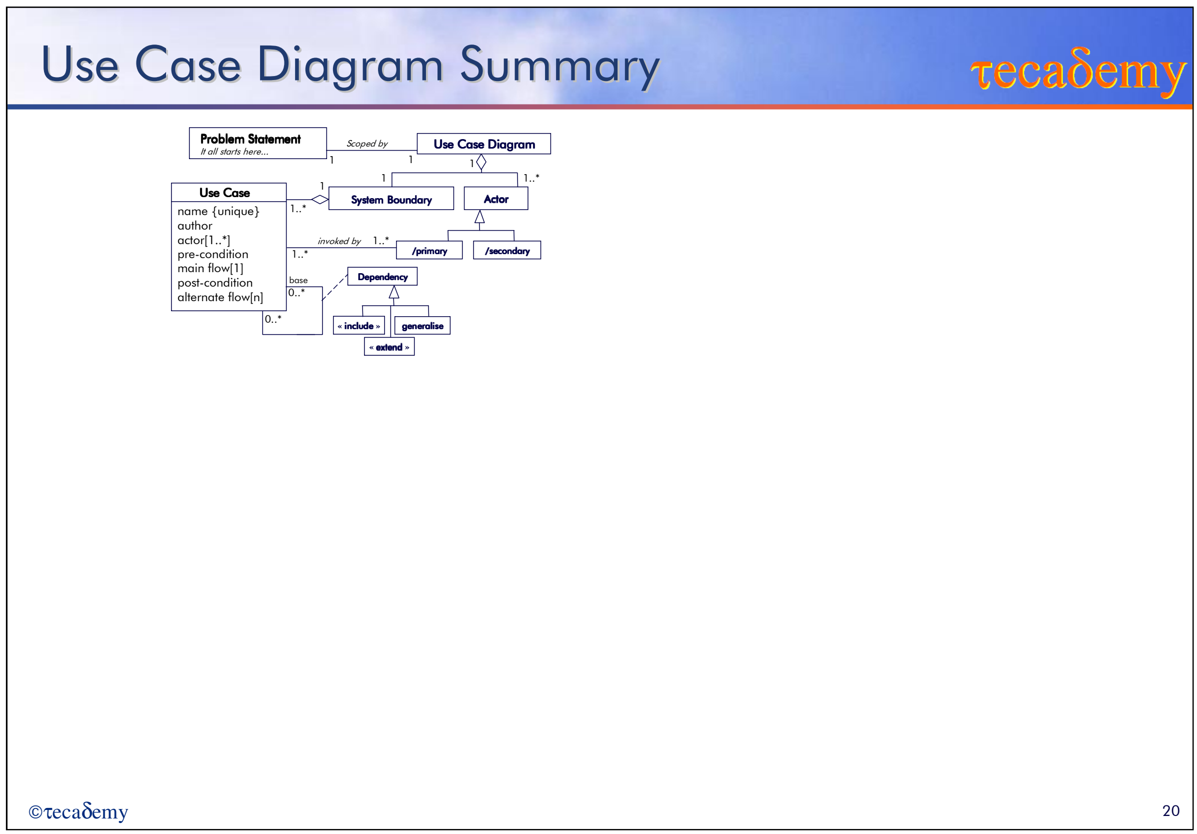
Once every cell has an entry, however, it can be shown that every possible combination of use cases and system states has been considered. The table could then be used as the basis of the system testing strategy. It could also form the basis of a contract, specifying the complete behaviour of a system to be delivered by a third-party supplier.

Summary

tecaδemy

- Various techniques can be used in the further analysis of a use case model
  - Packages
  - Actor generalisation/specialisation
  - Use case inclusion
  - Use case extensions
  - Use case generalisation
  - Pre- and postconditions
- Care should be taken not to over-engineer the model

©tecaδemy



Before we can start on use cases, we need a statement of the problem. The *Use Case Diagram* defines the scope of the problem. Use cases that are in scope are contained within the system boundary, actors are outside. An actor that invokes a use case is *primary* with respect to that use case; one that is contacted by a use case is *secondary* with respect to that use case. It is possible for an actor to be primary with respect to one use case and secondary with respect to another.

A use case can have relationships with other use cases – a use case can *include* or *extend* a *base* use case, or be a generalisation of more specific use cases.

Case Study - Exercise 6

τecaδemy

- Objective
  - Identify use case packages
  - Understand difference between includes, extends and generalisation
- Turn to and complete exercise 6 in the exercise booklet

©τecaδemy

Notes: