



Identifying Behaviour

Scenarios and Sequence Diagrams

Objectives

- Familiarisation with textual and visual scenarios
- Familiarisation with behavioural modelling techniques

Contents

- Writing scenarios
- Visually modelling scenarios using sequence diagrams
- Iteration

Example Scenario

For the use case: Make a Provisional Holiday Booking

Helen asks the system to reserve a place on the CityBreak in Barcelona on 18th October.
 The system uses the CBB1 code and the start date of 18th October to find the holiday package.
 The system temporarily holds a place on this holiday package.
 The system notifies Helen that a temporary booking is available.
 Helen accepts the place and supplies her name "Helen Backe" and her address details to the system.
 The system creates a temporary booking and assigns the place to Helen.
 A confirmation number is supplied to Helen by the system. ref. 26308

The above scenario shows a concrete example of a primary scenario, for the 'Make a Provisional Holiday Booking' use case.

Scenarios are written from the actor's point of view. Start the scenario description by stating who starts the scenario and what they are doing. Continue listing the steps until the goal is achieved and the use case ends.

It can be useful to try and construct each scenario as a sequence of clauses, each having an identifiable subject, verb and object (used here in its grammatical sense). This encourages thinking in terms of interacting instances – the client (subject) sending a message (verb) to the supplier (object). The subject and object nouns should be drawn, wherever possible, from the list of candidate objects. This may lead you to discover the need for additional objects – add them to the list. Using active verbs helps to reinforce the client-supplier roles. For example, in 'the boy kicks the ball', it is obvious who is doing what to whom; in 'the ball is kicked by the boy' it is the subject receiving the effect of the verb.

It is usually best to start by modelling a successful scenario. This may take a long time, but it is worth persevering because it makes the following scenarios easier to write. Errors and exceptions, for instance, invariably occur somewhere along the 'successful' path – just cut-and-paste up to that point and continue from there. It is also worth looking for 'sub-scenarios' – logical groupings of interactions – that can be used as cut-and-paste components with which to build other scenarios. The aim should be to identify a minimal set of sub-scenarios from which all possible scenarios can be constructed.

Actually playing out a scenario, with team members playing the part of objects and passing written messages and responses, can help generate these scenario 'scripts'. Don't forget to have someone record each successful interaction! A scenario is constrained to be a single pass through a use case, without any of the generalities allowed in a use case. It is a real example of what happens when using the system, whereas the use case is a generic description of the type of things that may happen. This instantiation of the use case ensures that we think more in the concrete than in the abstract, and think of a set of distinct possibilities that may occur.

Scenarios
tecademy

- ▶ **A scenario is an instance of a use case**
 - ▶ A real-world description of a Use Case in action
 - ▶ One of many possible paths through a Use Case
- ▶ **Scenarios encourage exploration of a use case**
 - ▶ Leads to more detailed consideration of a use case
 - ▶ Can uncover hidden requirements
 - ▶ Results in deeper understanding of problem
 - ▶ Can be described textually and graphically
- ▶ **Involved in future steps of process**
 - ▶ E.g. test cases, training manuals, user guides

©tecademy
4

A Scenario is an instance of a use case – one possible way of providing a use case’s behaviour. This implies a possibly infinite number of scenarios for each use case, since even two scenarios which look the same will differ in time and space. In practice, it is more useful to ignore this difference and concentrate on distinct patterns of behaviour. Even then, there could still be an intimidating number of possible scenarios. There will be at least one successful scenario (if there isn’t, the purpose of the use case should be questioned!), and at least one for each of the n errors and exceptions identified in the use case description. So the *minimum* number of scenarios for each use case will be $1 + n$; and you will find others.

During this activity you are likely to discover changes to use cases and actors. Don’t forget to record the changes in the Use Case documentation and glossary.

Scenarios are an excellent communication tool and very useful for discovering hidden requirements. Sometimes new alternatives or even whole use cases can be discovered from identifying and expanding scenarios.

Scenarios can be used to create sequence diagram and later in the project lifecycle can be used as test cases.

Example Scenario Titles tecademy

- For the use case: Make a Provisional Holiday Booking
 - Helen makes a booking successfully
 - Helen doesn't book, as there are no available places
 - Helen only manages a temporary booking
 - Helen makes the first booking, advised of possible cancellation
 - Helen makes a booking for a later date
 - Helen makes a booking for a cancelled holiday
 - Helen makes a booking for Paris
 - Homer tries to make a booking for last week
 - Homer tries to make a booking for a non-existent holiday
- Might be possible to find and re-use 'sub-scenarios' to reduce work

©tecademy 5

These are not scenarios, but the names of scenarios. A different scenario description would be written for each one. They represent a very small percentage of all the potential scenarios that could be generated for this use case. However, they may allow us to cover a large percentage of the behaviour being considered. That is, we don't need to write an infinite number of scenarios; rather, we need to find the smallest number that covers the Use Case adequately.

There is a strong chance that many of these scenarios will have segments in common. For example, many of the scenarios in the slide above will contain a segment that deals with retrieving course details. This segment could be usefully identified as a 'sub-scenario', which can then be re-used in all of the scenarios that require course-retrieval behaviour. Generally, when dealing with a use case that has a large number of potential scenarios, it can be a good idea to try to identify sub-scenarios from which other scenarios can be built. Because a very large number of scenarios can often be built from relatively few sub-scenarios, trying to identify them can be a useful investment.

Identifying and Writing

Identifying scenarios

- A use case can have a very large number of potential scenarios
- Start by listing 'normal' or 'sunny day' scenarios
- Look for other ways of doing the same thing
- Look for errors
- Look for exceptions

Writing scenarios

- 'Sunny day' scenario first
- Use specific values to illustrate all data
- Keep logical - do not include user interface
- Keep simple - use candidate classes and business vocabulary
- Subject-verb-object on each line
- Users should be involved as much as possible

Identifying scenarios is a similar activity to finding test cases. As in testing, we would like to be able to write the smallest number of scenarios that gives us the greatest depth of understanding.

The earliest scenarios written for a use case need to follow the use case closely, starting with the 'sunny day' scenario – the one where nothing goes wrong. Every use case *must* have at least one of these – otherwise the purpose of the use case has to be in question! This is likely to be the most difficult to write, but it is worth persevering – most of the alternate and exception flows will branch off from the 'sunny day' path and it will therefore act as a framework for the others.

To continue the process, look for the other most common scenarios of this use case; the ones the actor will most often pursue, and any that are easy to understand. Only when a sufficiently-detailed knowledge of the main thrust of the use case is acquired, should less-frequent or more-complex scenarios be created.

Also, expect the unexpected; the actors might do unexpected things, and it is the analyst's job to expect them. Any of the objects inside the system might also behave badly or erratically, probably due to an error, possibly from some malicious intent. It is important to recognise these potential irregularities as early as possible, otherwise they will be found by users, with potentially embarrassing consequences.

There are two distinct steps in creating scenarios. The first is to identify the scenarios, and the second is to write full length descriptions. By checking the model in the light of new scenario the impact on the model can be examined. It is often found at this point that the use case is not very well founded, and it may be split into differing use cases, or merged with an existing one. Likewise functional details may need clarifying or changing in consultation with a business domain expert.

The best people to write scenarios are those who actually do the job being described. Where this isn't possible, they should at least be able to verify or correct the content.

As we will see later the approach taken in the Unified Process is to work in an iterative fashion prioritising use cases and alternatives based on risk, both from business and technical perspectives.

Sub-scenarios

- Look for logical 'chunks' of behaviour
 - e.g. scenario fragments that can be cut-and-pasted into other scenarios.
- «includes» and «extends» use cases lead to sub-scenarios...
 - ... and vice versa!
- Sub-scenarios provide a 'component-based' approach to writing scenarios...
 - ... and can reduce the amount of work considerably!

It can be more effective to look for sub-scenarios from which other scenarios can be built than to try and write all possible scenarios!

An example of a sub-scenario in , say, a 'Get money from an ATM' use case might be 'Check PIN'. 'Check PIN' wouldn't qualify as a use case in its own right because it never occurs on its own, only within the context of goal-oriented use cases such as... well, what do *you* use an ATM for? Bet you've never used it just to check you remember your PIN! But there is value in modelling 'Check PIN' as a sub-scenario, because it *can* occur as part of many other scenarios.

Another example follows.

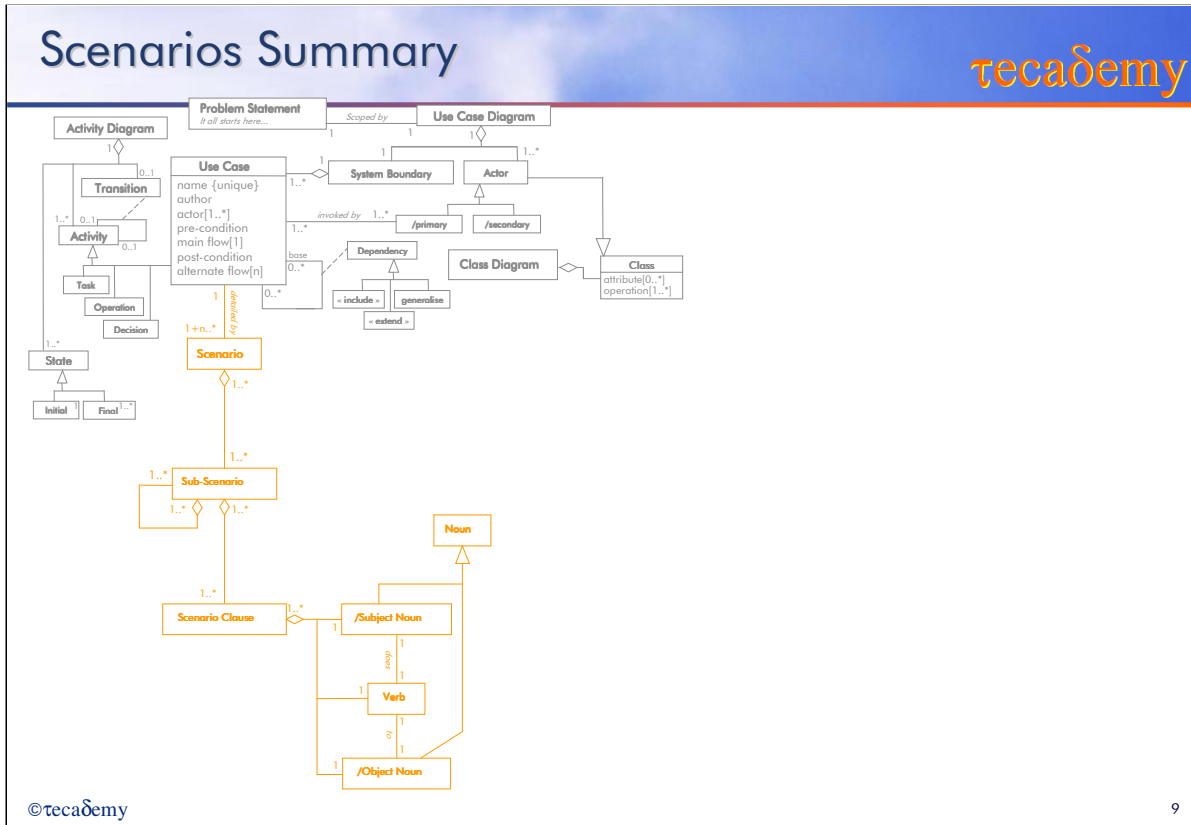
Sub-scenarios - Example

▶ For the use case: Make a Provisional Holiday Booking

▶ Possible sub-scenarios:

- ▶ Record holiday-maker's details
- ▶ Check availability of holiday package
- ▶ Record payment information
- ▶ Book holiday
- ▶ Book flight
- ▶ Book rental car
- ▶ Sell travel insurance

None of these examples is a full or complete scenario on its own (although some *could* be), but they can be mixed-and-matched to construct many possible alternate scenarios for the 'Make a Provisional Holiday Booking' use case. It is certainly less work to identify and write the sub-scenarios than it is to write all the possible combinations individually!



A *scenario* is a description of one instance of a use case in action, using a real example. There will be at least one successful scenario, possibly more; and several (*n*) error or exception scenarios. Each scenario is written as a sequence of subject-verb-object clauses. It is good practice to identify sub-scenarios – a sub-scenario being a piece of behaviour that could be repeated in other scenarios – as this can reduce the total number of scenarios that need to be written. A sub-scenario could map to an inclusion or extension use case, or could just be a piece of behaviour repeated in lots of scenarios.

Case Study - Exercise 8

- Objective

- To practice writing Scenarios.

- Turn to and complete exercise 8 in the exercise booklet.

Sequence Diagram

- A sequence diagram is a graphical scenario
 - Text might be easier for lay readers
 - Graphical constructs can be more precise, and become more-easily read by the initiated

- Creating the sequence diagram ensures that the objects and messages involved are formally identified and highlighted
 - Removes some of the vagaries of natural language
 - Moves towards a common vocabulary

So far in the course we have covered use cases, scenarios and the basics of class diagrams. These are not in themselves enough to understand the problem domain. This chapter builds this understanding; that is, it provides a more-detailed functional view of the system, a better description of our domain terms and a model of how these domain items interact dynamically.

Sequence diagrams essentially contain the same information as scenarios. Normally, however, the more-formal layout of the sequence diagram causes us to go into more detail than in the scenario, and we find that the scenario was incorrect or incomplete. We use sequence diagrams to uncover ambiguities and omissions in the scenarios.

Many people dislike writing verbose textual scenarios and go straight to sequence diagrams.

The sequence diagrams also provide a better correlation between this dynamic view of the system and the static view held in the class diagram. While it is easy to leave a slang term or a pronoun (or any other strange grammatical terms people use) in the scenario, in a sequence diagram the domain terms have pride of place, and won't be allowed to be different from the domain vocabulary in the class diagram for long.

From Scenario to Sequence Diagram

Scenario

obj1 says this to obj2
obj2 says that to obj1
obj1 creates obj3
obj1 asks obj2 to do something
obj2 asks obj3 to help
obj3 checks itself
obj3 says ok! to obj2
obj2 says finished! to obj1
obj1 says die! to obj3

Sequence Diagram

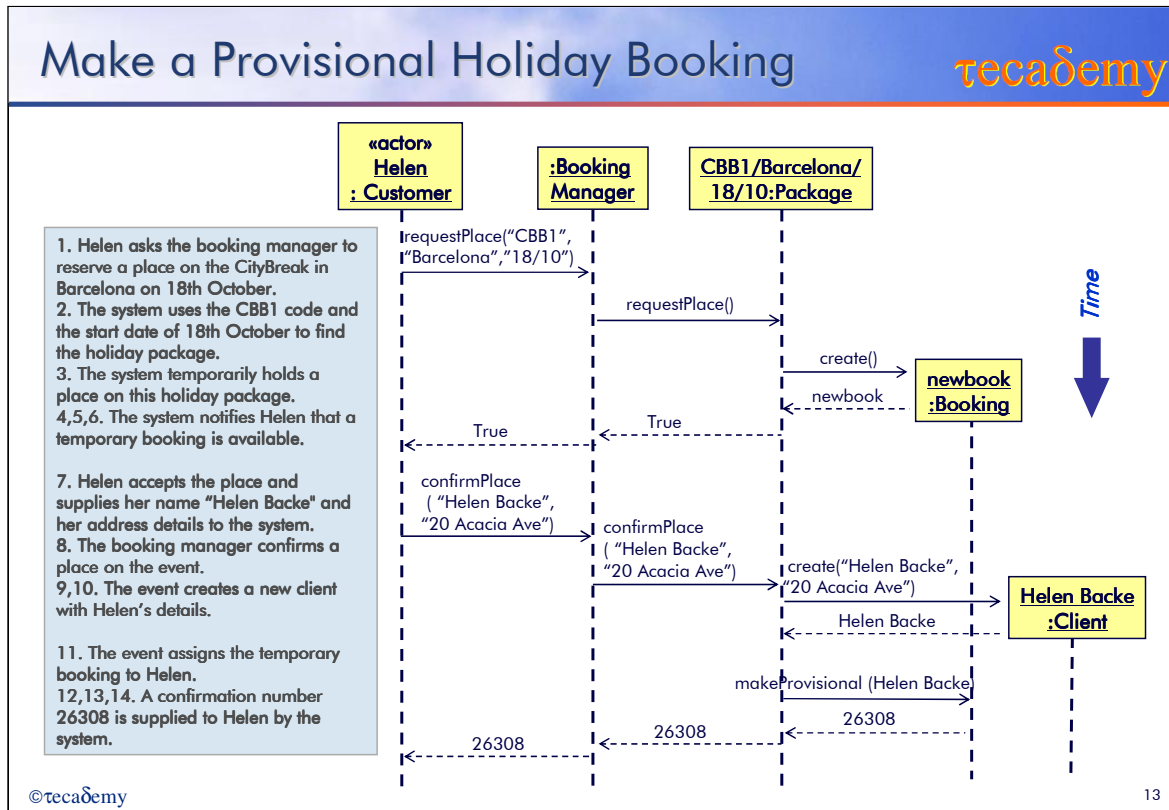
- Subject noun becomes 'client' object
- Verb becomes message
- Object noun becomes 'supplier' object

©tecademy
12

So, sequence diagrams give us another way to represent scenarios. Transforming a scenario into a sequence diagram is relatively straightforward if the scenario has been written in the line-by-line, subject-verb-object format suggested earlier. The use of active (rather than passive) verbs makes it easier to see which way the message is being sent (compare 'Janet kicks the dog' with 'The dog is kicked by Janet'.)

Notationally, a sequence diagram shows each object with its time-line running down the page. Messages (from the verbs) are shown as arrows pointing from the sender (subject noun) to the receiver (object noun). Objects appear at the point at which they are created – the 'create' message points to the object icon. When an object is destroyed, the object's time-line ends with an 'X'.

Objects that are created and destroyed within the scenario/sequence diagram (for example, obj3 in the slide above) are known as *transient* objects. Objects that are not transient are *persistent*, which means they need to be stored between scenarios. To be thorough, you should try to account for the creation and destruction of every object in your models – either by modelling these events as additional scenarios, or by identifying another system where these events take place.



The diagram above shows a more realistic example of a sequence diagram.

The instances Helen, CBB1/Barcelona/18/10, newbook and Helen Backe are clearly visible, as are their classes: **Customer**, **Package**, **Booking** and **Client**. The **BookingManager** object is an example of an anonymous object – a name could be given if needed. The order of the objects is not significant. The order from top to bottom is significant, time flows from top to bottom, as messages and returns pass between the objects.

The text «actor» is a stereotype, indicating that Helen is the actor; a real, warm human being, as opposed to a software object that contains information about Helen (the :Client object). By convention, sequence diagrams begin with the actor.

Sequence Diagrams

- ▶ Objects, not classes
- ▶ Normally at least one actor
- ▶ Named parameters and objects encouraged
- ▶ Pre-existing objects at the top
- ▶ Newly-created objects at their point of creation
- ▶ Object names used as parameters and returns
- ▶ May discover new classes

Sequence diagrams contain objects, not classes (unlike a class diagram, which contains classes that signify the existence of objects). If more than one object of a particular class needs to appear in a sequence diagram, multiple headings and vertical lines are required to show each object distinctly. A scenario describing a transfer of funds from one bank account to another would have both bank accounts as individual objects in the sequence diagram.

The objects for a sequence diagram will come from the nouns in the scenario, and also from any class diagram so far created. Some scenario nouns may be identical to the class diagram classes, some may be similar (synonyms), which need to be recognised as the same, and others will be new ones, which will need to be added to the class diagram.

Most scenarios are initiated by a primary actor. Some scenarios have secondary actors, who are involved, but do not initiate (at least not in scenarios with other primaries). Some scenarios have no actors, but are initiated by the system itself, usually when some threshold has been passed, such as a time point, or the thousandth customer.

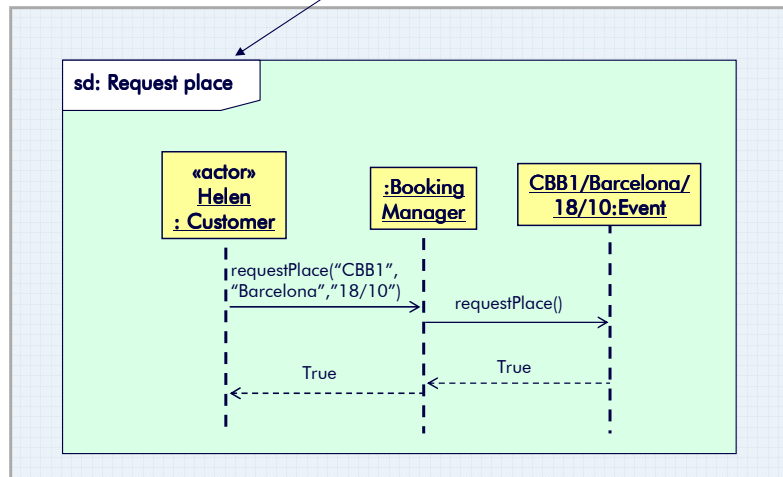
Normally there will be at least one actor on the sequence diagram. More actors may also be shown. If there are multiple actors participating in the scenario these too should be shown on the use case sequence diagram. Secondary actors are often shown on the right hand side.

Newly created objects are rendered further down the page.

Also notice how the names of the object are used in parameter passing and returns to show objects being passed within the system. In the example `makeProvisional` passes the `JimWear` object as a parameter.

Labelling Sequence Diagrams

- Sequence diagrams should be labelled for reference:

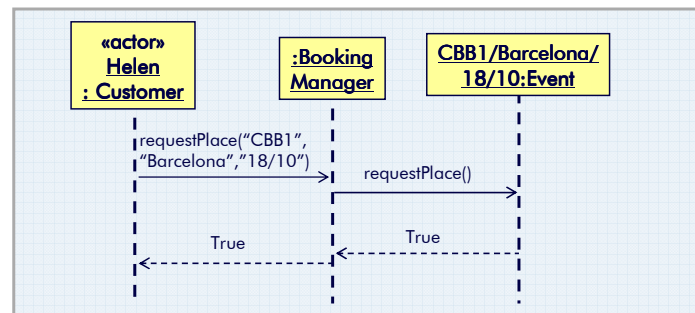


Individual sequence diagrams, whether for whole scenarios or scenario fragments, can be labelled as in the slide above. The whole sequence diagram is enclosed in a rectangle, which has a pentagonal box (or a rectangle with a corner cut off, if you prefer) in the top left-hand corner bearing the name of the diagram.

Sequence diagram names, and those of the related scenario and communication diagram (of which more later) are prefixed by the letters 'sd' followed by a colon.

How do objects communicate ?

- ▶ Objects communicate by sending or replying to messages
 - ▶ Messages and replies are events – both are captured
 - ▶ Object dialogue reduced with parameters
 - ▶ Try consistently to name messages as imperative actions on the receiving object
 - ▶ Returns, if there are any, shown as dashed line
 - ▶ Returns must go back to the sender of the message



One of the primary reasons for sequence diagrams is to identify operations on classes. It is rather difficult to do this directly from the static class diagram, as the class diagram does not contain any context for using the objects. Use cases, scenarios and sequence diagrams attack the problem from a dynamic (rather than static) point of view, asking the questions, "What does the user want to do with the system?", "Which objects are involved?" and "What behaviour must be present inside the system to provide what the user needs, and how can that behaviour be apportioned to the identified objects?". Each user interaction must be looked at from the user's perspective, i.e. "What do I want the system to do?", and from the objects' perspective, i.e. "How do I do that?".

Messages and replies are communications between objects. From the sender's point of view, a message requests the recipient to do something, or tells the recipient that something has happened. From the receiver's point of view, it is an order that must be obeyed, or a notification. Messages can carry information with them as parameters to reduce dialogue between objects. Replies do not carry parameters and are therefore shown without parentheses; they also can be shown using dashed arrows. Not all messages need replies – however, if there is one, it should be captured on the diagram. Messages and replies are regarded as *events* by the receiving object.


Messages invoke operations on the receiving object. Message names, therefore, should be stated as imperatives – they will become the names of the operations they are invoking.

Note a message is a generic kind of communication between objects. This may in fact be implemented in many ways e.g. distributed objects, RPC, normal procedure call. At this stage we only concern ourselves with the communication itself.


As stated above, each message may have information associated with it, but beware of just showing data flow as the reason why information is being passed is more important.

Interactions Types
tecademy


➤ Stick arrows are typically used before a decision is made

Filled Arrow 

- » Synchronous, nested flow of control
- » Sender cannot send further messages until reply is received
- » Sender expects a reply

Dashed Arrow 

- » Return from a synchronous call

Half Arrow 

- » Asynchronous call
- » Sender does not expect reply
- » Sender may send further messages

©tecademy
17

Strictly speaking, the arrows on a sequence diagram represent *events*. These events might be message events or return events. Not all messages in an interaction need to be of the same type; some objects may interact synchronously whilst others may interact asynchronously. This might happen when message are sent to a separate, independently executing components.

Filled Arrow. The sender loses control until the receiver finishes handling the message at which point the sender regains control. This is shown optionally with a return arrow:

Return. This is not a message but a return from an earlier message. It unblocks a synchronous send.

Half Arrow. This message does not expect a reply. The sender may stay active and may send further messages. This arrow is sometimes depicted with a filled arrow head.

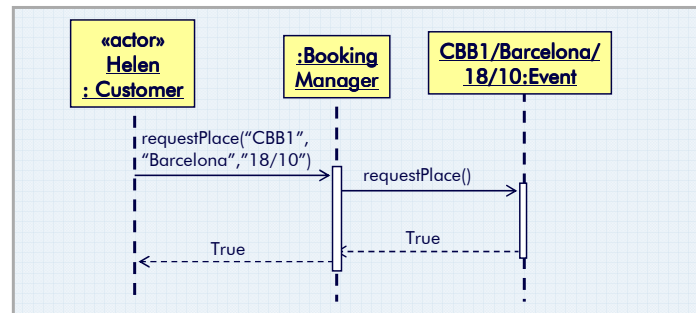
In a 'normal' procedural model, generally the first two arrow-types are used. UML has defined a number of variants of simple message passing to allow concurrent systems to be described.

By convention, message events are shown with parenthesis (which might carry parameters) while return events, which are simple values, variables or individual objects, do not.

This slide describes the standard UML definitions for object interaction. Conventions do differ so you may see variants, as with all aspects of UML further variants can be added using stereotypes to extend the core language.

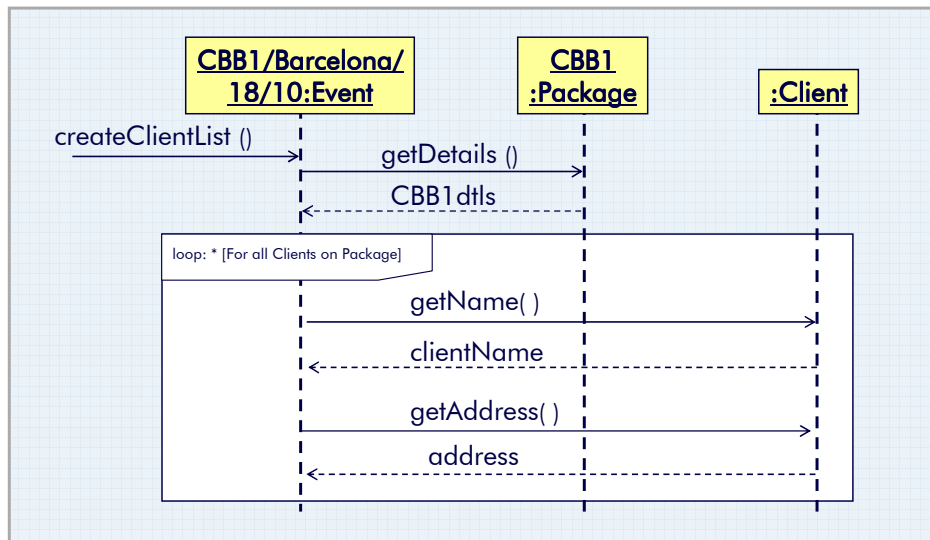
Control Regions

- Control Regions shows the period of time an object is performing an action
- Corresponds to an operation performed by object



A control region is shown as a fat, empty rectangle on an object's timeline. It indicates that this object is currently active within the system. A nested control region can be shown when an object sends a message and waits for its return. A nested control region can appear on the same object, in which case it is shown slightly to the side. This is often used to show the decomposition of complex operations into lower-level, more simple ones. There should not be control regions on the actor, as they are by definition, out of our control.

- Box can be used to visually group messages

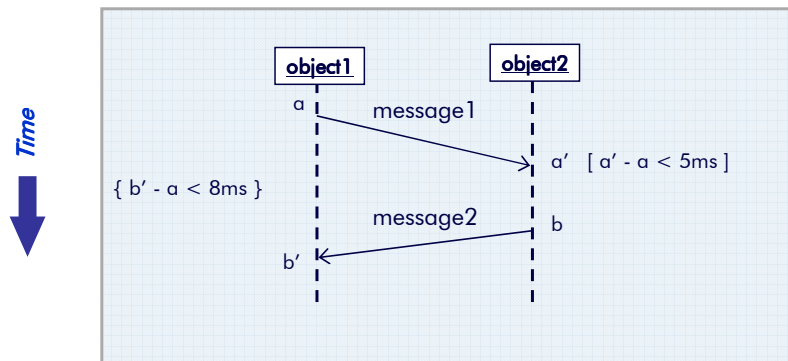


An enclosure may be drawn around a loop of messages. Note as with this example, an iteration box means repeat this sequence of messages which could be the subset of a set of messages from an original source message.

If just one message is repeatedly sent it can be prefixed with a *. All iterations should be accompanied with an iteration clause shown in square brackets, where the iteration clause determines the condition under which the message or sequence of messages are repeated.

Timing

- Message arrows may be slanted to show passage of time between send and receive
- Labels may be used to capture timing points
- Guards and constraints can contain timing marks

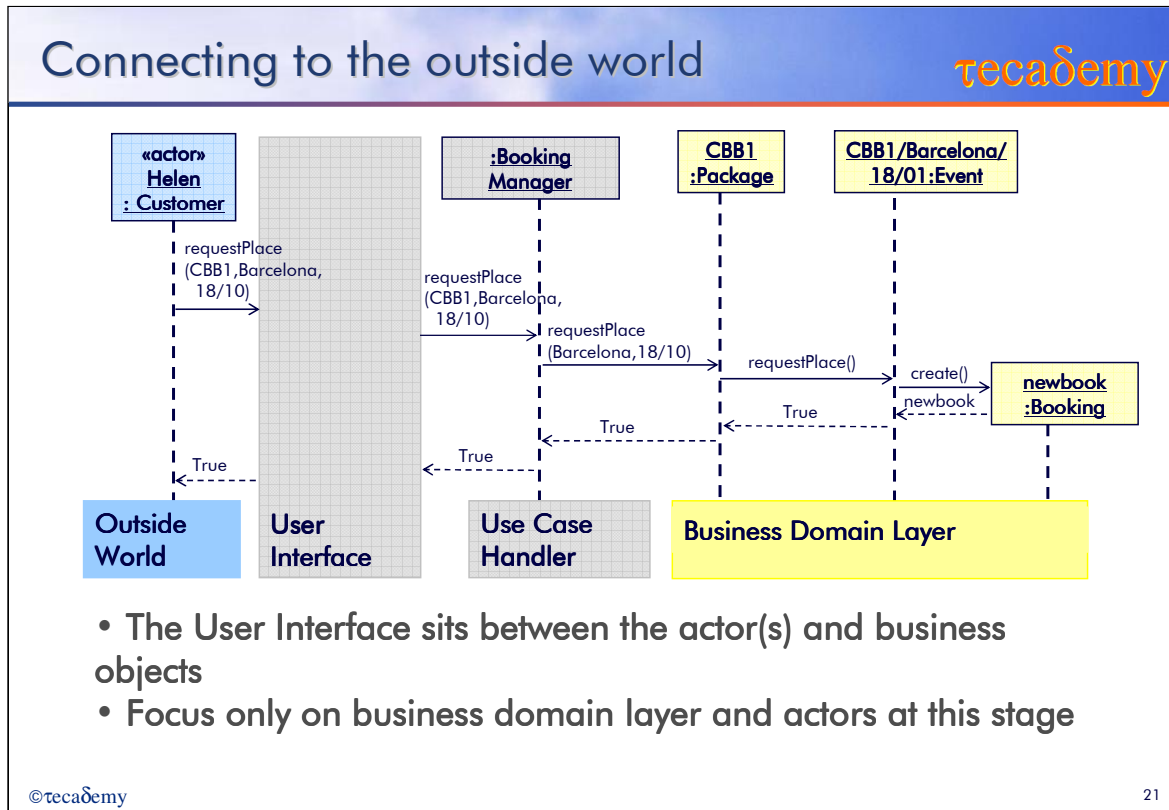


Messages can be shown as taking time to arrive by slanting the message arrow. Be sure to slant with the flow of time, so as to avoid any paradoxes caused by time-travel into the past...

If required, the send and receive times can be labelled, then used in guard conditions and timing constraints.

In the diagram above, the guard condition ' $[a' - a < 5ms]$ ' specifies that message2 is sent **because** a' was received less than 5ms after a was sent. (This is a sequence diagram for a single scenario, so we know message2 was sent in this case.)

The constraint ' $\{ b' - a < 8ms \}$ ' specifies a requirement that the system should be built such that message2 will be received less than 8ms after message1 was sent.



The models so far have shown direct access between an actor and an object. In the full system the actor will interact with the user interface which itself will be composed of many objects. One way to bridge the gap between the user interface objects and the business objects is to create a handler object perhaps, one for each use case. This object has the responsibility of passing messages onto business domain objects acting like a bridge. It does not have functionality of its own and acts purely as an agent passing messages on.

For the purpose of this course we focus on modelling the business domain layer.

Creating Sequence Diagrams

- ▶ **Sequence diagrams need to get to the right level**
 - ▶ Try to avoid providing the ultimate solution
 - ▶ Work iteratively through trial and error
- ▶ **Translate from a scenario**
- ▶ **Objects**
 - ▶ Nouns of the scenario (Classes from class diagram/list)
- ▶ **Messages**
 - ▶ Verbs of the scenario
- ▶ **Try to capture all the problem information**
 - ▶ Use it to name objects and parameters

A sequence diagram generally has a one-to-one relationship with a scenario. They can thus be constructed by a translation of the scenario into this more formal and graphical format. However, most people find sequence diagrams relatively natural, and often omit the phase of creating scenarios. Having observed this, we still maintain that it is usually worthwhile to create some textual scenarios for each use case. The benefits are that they allow more diversity, both in terms of language used (synonyms, etc.), and the range of behaviour considered.

Neither scenarios nor sequence diagrams give much help in identifying parameters and return types; however, they do need to be found.

Model builders need to find a balance between having too little detail, which says nothing new, and having too much detail, which, while saying what needs to be said, is unreadable by anyone except the author. Finding the right level for sequence diagrams is difficult, but hopefully the above guidelines will help.

An operation should be considered correctly detailed if you could outline an implementation, showing what information it needs, without recourse to detailed operations on other objects.

If you already have the code, you have gone too far.

If you have no idea what is needed in terms of further operations or information, you have not gone far enough.

Also remember that interaction should always be at the logical level; never model menu selections or mouse clicks, or even filling in a field on a form.

General Behavioural Heuristics

- Model real world where possible
- Assign responsibilities to fit class intent
- Distribute functionality as uniformly as possible
- Do not model an actor as an object
- Keep related behaviour and data together
- Honour encapsulation

When sending a message to an object it will then be responsible for some action relating to that message. In effect we are assigning responsibilities when creating a sequence diagram. Usually assigning responsibilities is reasonably obvious - what is the real world thing that would be activated.

The descriptions of the classes created in the last step can be used to assign responsibilities. If the intent is clear this is likely to facilitate distributing the functionality horizontally across the system. If behaviour is required that doesn't relate to the intent of any of the existing classes this may be an indication another class is required.

In an object-oriented system we try to distribute the intelligence among different types of objects rather than keeping it in one place. This will help facilitate a system that is resistant to change in the future.

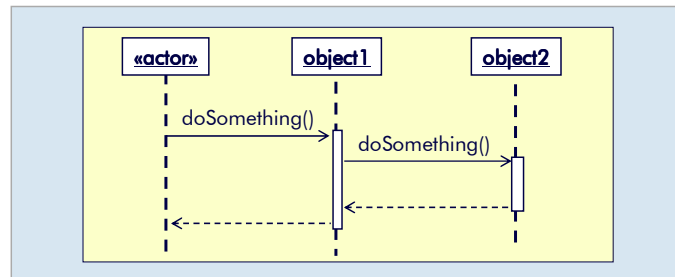
Try to make the objects you create intelligent. Be suspicious if they only seem to hold data or if they have behaviour and get all the data required from somewhere else.

If an object sends messages into the model but never receives any messages question whether you have modelled an actor as an object. E.g. an administrator is acting on behalf of the customer to join library. This sequence diagram should not start with the actor customer sending a join message to the administrator object that then send a join message to the library object. The administrator is outside the system.

Try to avoid retrieving data out of objects, and then performing some calculation on that data and then sending a message to set it back again. In this situation ask yourself why isn't this object doing this work for me? Encapsulation means that we do not access internal data of an object even via a get operation unless it's absolutely necessary.

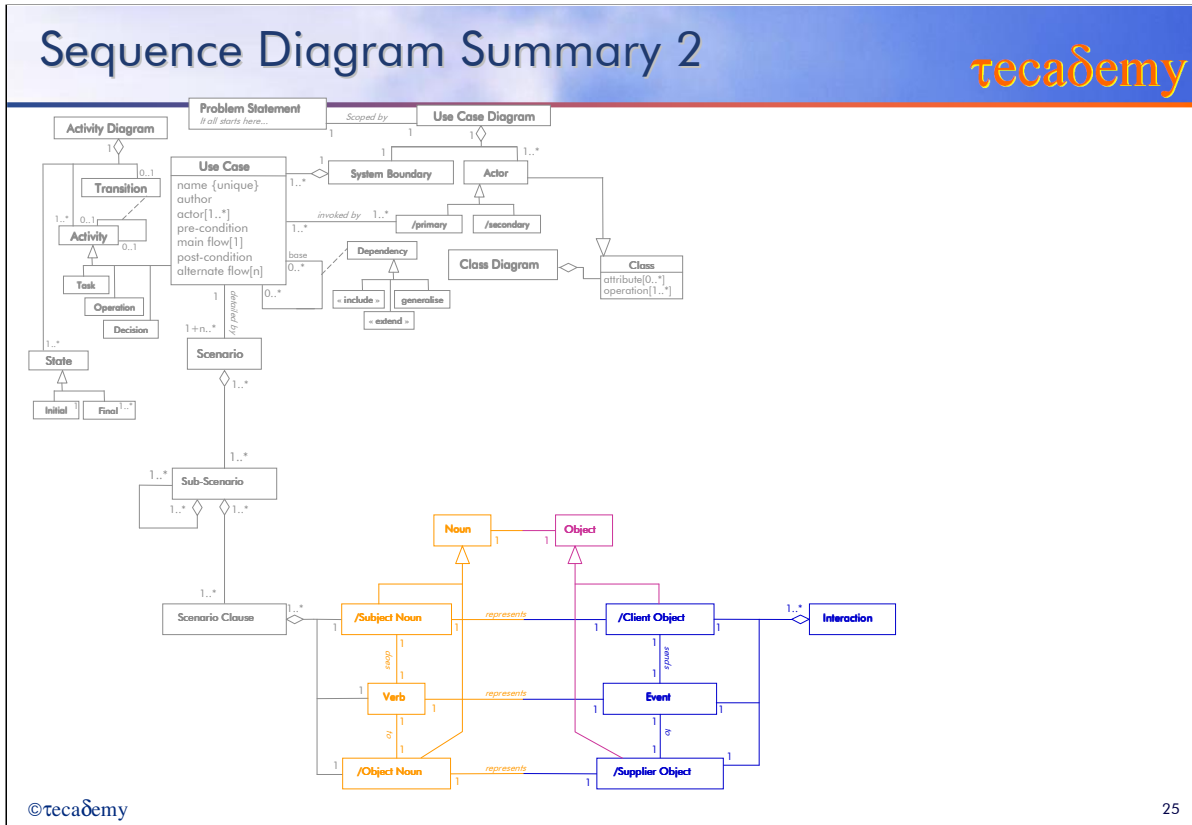
Sequence Diagram Summary 1

- ▶ A sequence diagram is a graphical scenario that shows object interaction in a time based sequence

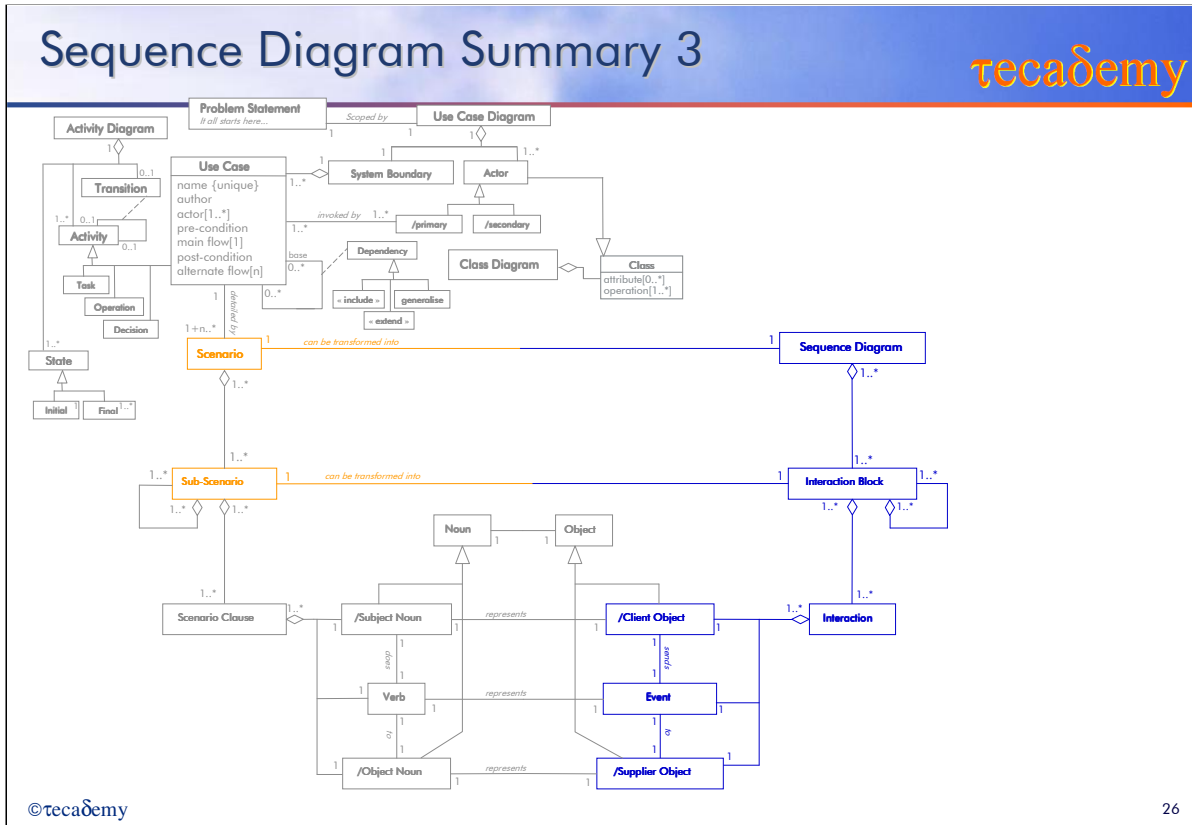


- ▶ Graphical constructs tend to be more precise than text
- ▶ Creating the sequence diagram ensures that the objects and messages involved are formally identified and highlighted
- ▶ Identify relationships, operations and new classes

- ▶ As a general guideline, try to keep the diagrams simple



The subject noun does something 'verby' to the object noun (using 'object' in the grammatical sense). If we map nouns to *objects*, the subject noun represents the *client* object, the object noun the *supplier*, and the verb (preferably in the active rather than passive form) shows a message passing from client to supplier. This represents a single interaction between two objects.



And just as a scenario clause can map to a single object interaction, a scenario can be represented as one sequence diagram. They are simply different representations of the same information – one textual, one graphical.

Case Study - Exercise 9

- Objective
 - To practice drawing Sequence Diagrams

- Turn to and complete exercise 9 in the exercise booklet

Scenarios and Sequence Diagrams

Notes: